容器与算法

主讲教师： 沈艳艳

Vector and Free Store

- Vector is not just the most useful standard container.
- It provides examples of the most important implementation techniques.

Vector

- Vector is the most useful container
  - Simple
  - Compactly stores elements of a given type
  - Efficient access
  - Expands to hold any number of elements
  - Optionally range-checked access
- How is that done?
  - That is, how is vector implemented?
    - We'll answer that gradually, feature after feature
- Vector is the default container
  - Prefer vector for storing elements unless there's a good reason not to

Building from the ground up

- The hardware provides memory and addresses
  - Low level
  - Untyped
  - Fixed-sized chunks of memory
  - No checking
  - As fast as the hardware architects can make it
- The application builder needs something like a vector
  - Higher-level operations
  - Type checked
  - Size varies (as we get more data)
  - Run-time range checking
  - Close to optimally fast

Building from the ground up

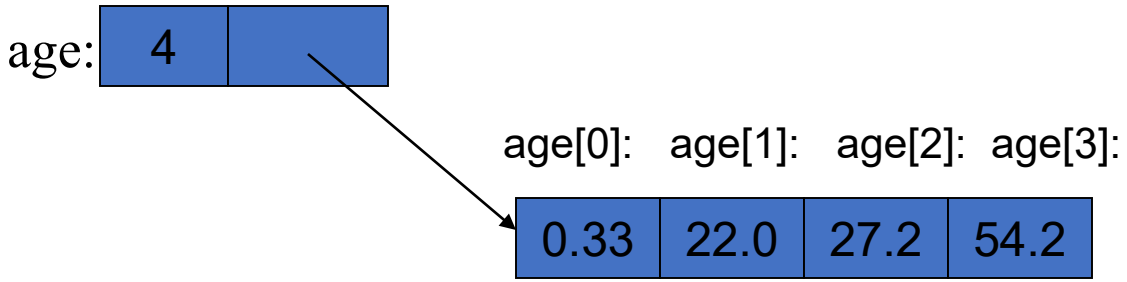
- At the lowest level, close to the hardware, life's simple and brutal
  - You have to program everything yourself
  - You have no type checking to help you
  - Run-time errors are found when data is corrupted or the program crashes
- We want to get to a higher level as quickly as we can
  - To become productive and reliable
  - To use a language "fit for humans"
- Chapters 17-19 basically show all the steps needed
  - The alternative to understanding is to believe in "magic"
  - The techniques for building vector are the ones underlying all higher-level work with data structures

Vector

- A **vector**
  - Can hold an arbitrary number of elements
    - Up to whatever physical memory and the operating system can handle
  - That number can vary over time
    - E.g. by using **push_back()**
  - Example

    vector<double> age(4);

    age[0]=.33; age[1]=22.0; age[2]=27.2; age[3]=54.2;



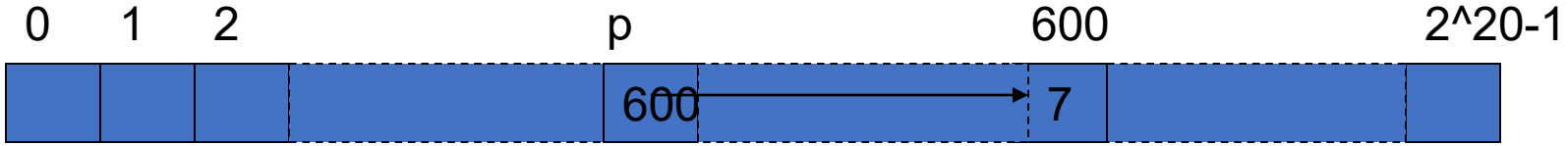
Vector

```
// a very simplified vector of doubles (like vector<double>):
class vector {
   int sz;              // the number of elements ("the size")
   double* elem; // pointer to the first element
public:
   vector(int s);                        // constructor: allocate s elements,
                                         // let elem point to them,
                                         // store s in sz
   int size() const { return sz; }       // the current size
};
```

- * means "pointer to" so **double*** is a "pointer to **double**"
  - What is a "pointer"?
  - How do we make a pointer "point to" elements?
  - How do we "allocate" elements?

Pointer values

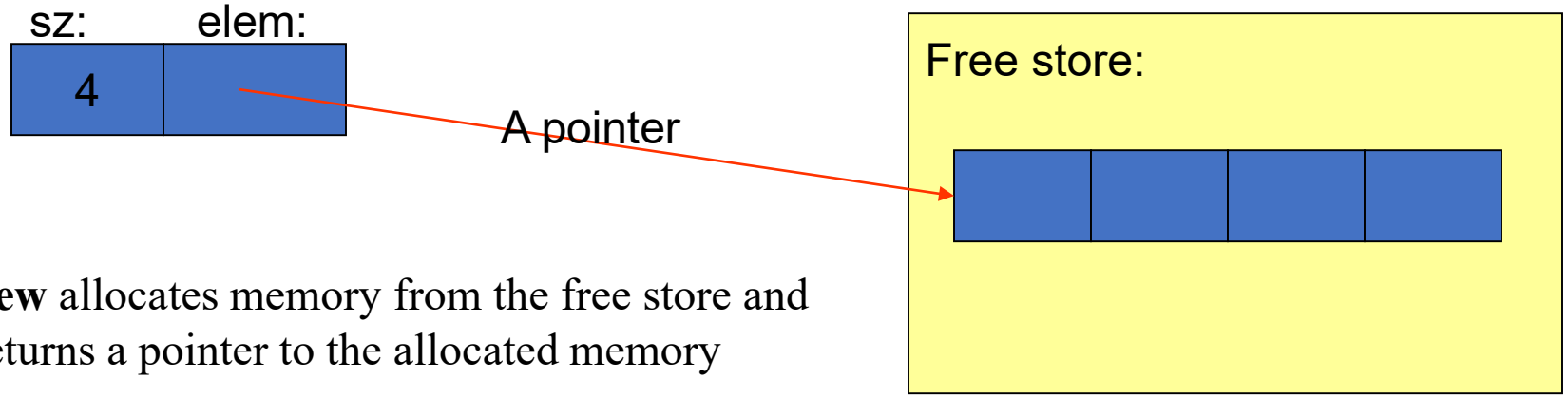
- Pointer values are memory addresses
  - Think of them as a kind of integer values
  - The first byte of memory is 0, the next 1, and so on
  - A pointer **p** can hold the address of a memory location



  - A pointer points to an object of a given type
    - E.g. a **double*** points to a **double**, not to a **string**
  - A pointer's type determines how the memory referred to by the pointer's value is used
    - E.g. what a **double*** points to can be added but not, say, concatenated

Vector (constructor)

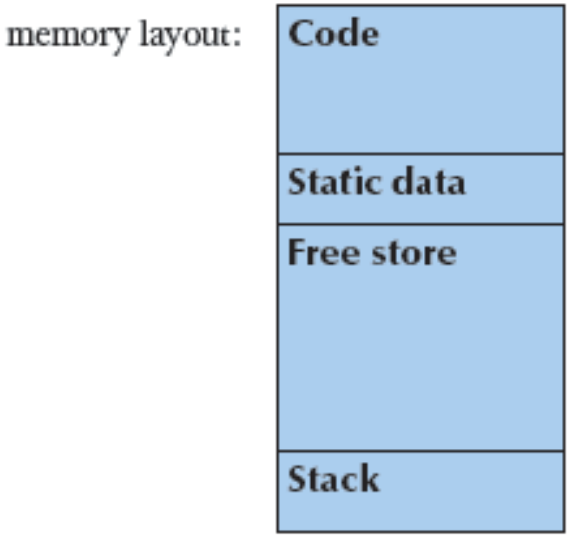
```
vector::vector(int s)          // vector's constructor
    :sz(s),                    // store the size s in sz
     elem(new double[s])       // allocate s doubles on the free store
                               // store a pointer to those doubles in elem
{
}
// Note: new does not initialize elements (but the standard vector does)
```



new allocates memory from the free store and returns a pointer to the allocated memory

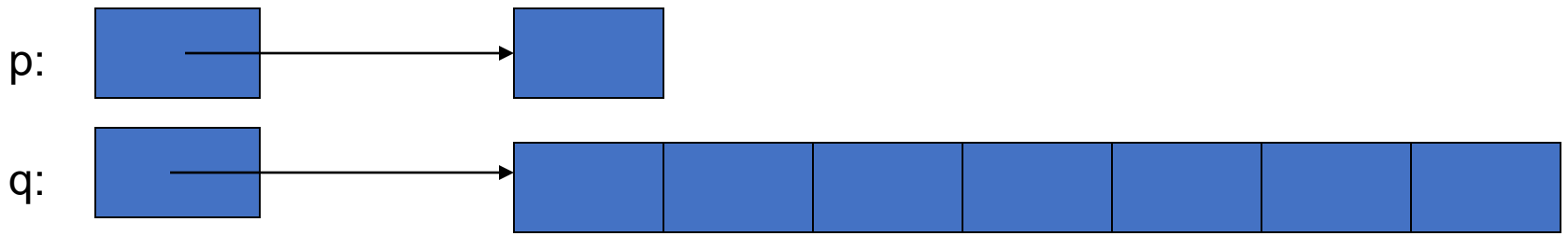
The computer's memory

- As a program sees it
  - Local variables "live on the stack"
  - Global variables are "static data"
  - The executable code is in "the code section"

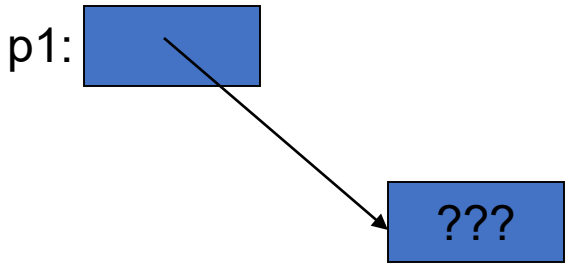


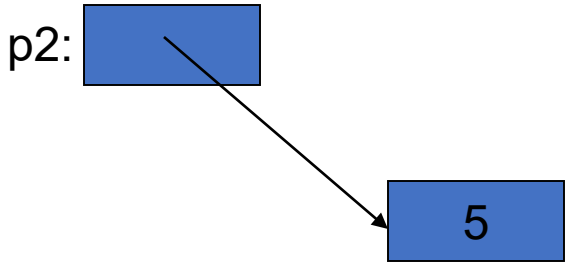
The free store
(sometimes called "the heap")

- You request memory “to be allocated” “on the free store” by the **new** operator
  - The **new** operator returns a pointer to the allocated memory
  - A pointer is the address of the first byte of the memory
  - For example
    - **int* p = new int;** // *allocate one uninitialized* ***int***
      // ***int**** *means “pointer to* ***int****”*
    - **int* q = new int[7];** // *allocate seven uninitialized* ***int****s*
      // *“an array of 7* ***int****s”*
    - **double* pd = new double[n];** // *allocate* ***n*** *uninitialized* ***double****s*
  - A pointer points to an object of its specified type
  - A pointer does ***not*** know how many elements it points to



Access

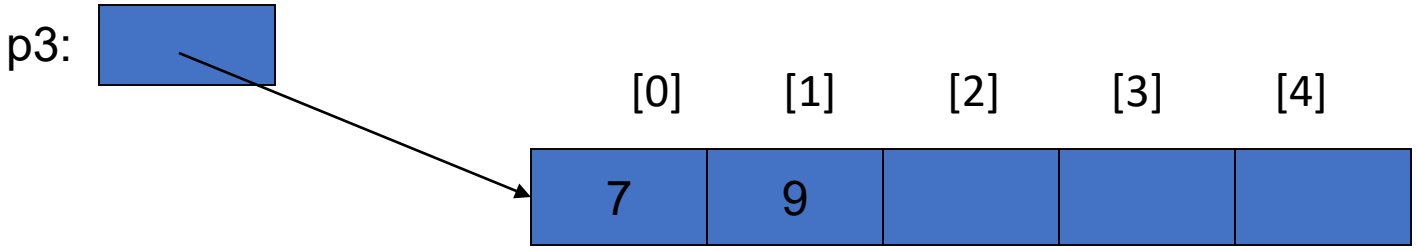




- Individual elements

```
int* p1 = new int;          // get (allocate) a new uninitialized int
int* p2 = new int(5);       // get a new int initialized to 5

int x = *p2;                // get/read the value pointed to by p2
                            // (or "get the contents of what p2 points to")
                            // in this case, the integer 5
int y = *p1;                // undefined: y gets an undefined value; don't do that
```

Access



- Arrays (sequences of elements)

```
int* p3 = new int[5];  // get (allocate) 5 ints
                       // array elements are numbered [0], [1], [2], …

p3[0] = 7;             // write to ("set") the 1st element  of p3
p3[1] = 9;

int x2 = p3[1];        // get the value of the 2nd element of p3

int x3 = *p3;          // we can also use the dereference operator * for an array
                       // *p3 means p3[0]  (and vice versa)
```

Why use free store?

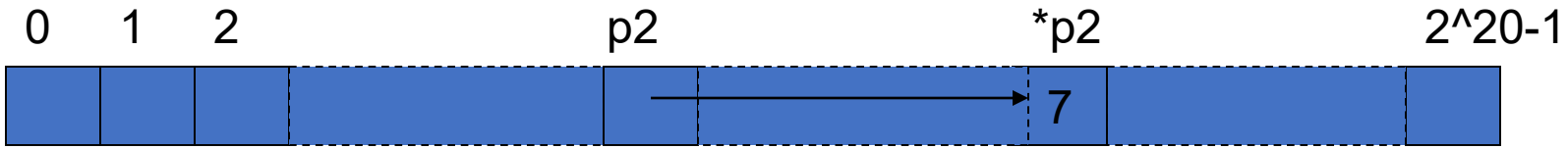
- To allocate objects that have to outlive the function that creates them:
  - For example

```
double* make(int n)   // allocate n doubles
{
    return new double[n];
}
```

  - Another example: vector's constructor

Pointer values

- Pointer values are memory addresses
  - Think of them as a kind of integer values
  - The first byte of memory is 0, the next 1, and so on

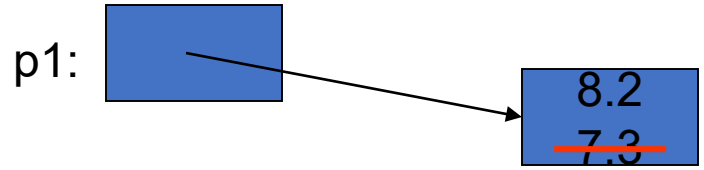


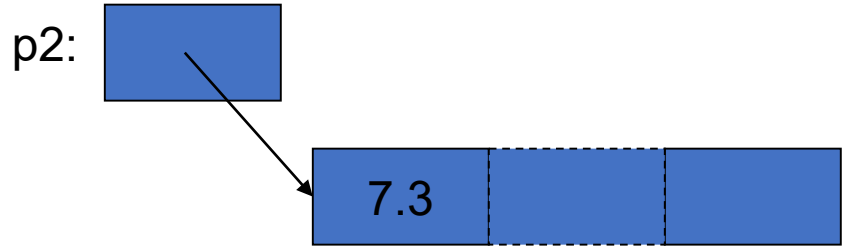

```
// you can see a pointer value (but you rarely need/want to):
int* p1 = new int(7);          // allocate an int and initialize it to 7
double* p2 = new double(7);    // allocate a double and initialize it to 7.0
cout << "p1==" << p1 << " *p1==" << *p1 << "\n";  // p1==??? *p1==c
cout << "p2==" << p2 << " *p2==" << *p2 << "\n";  // p2==??? *p2=7
```

Access

- A pointer does **not** know the number of elements that it's pointing to (only the address of the first element)

```
double* p1 = new double;
*p1 = 7.3;          // ok
p1[0] = 8.2;        // ok
p1[17] = 9.4;       // ouch! Undetected error
p1[-4] = 2.4;       // ouch! Another undetected error

double* p2 = new double[100];
*p2 = 7.3;          // ok
p2[17] = 9.4;       // ok
p2[-4] = 2.4;       // ouch! Undetected error
```

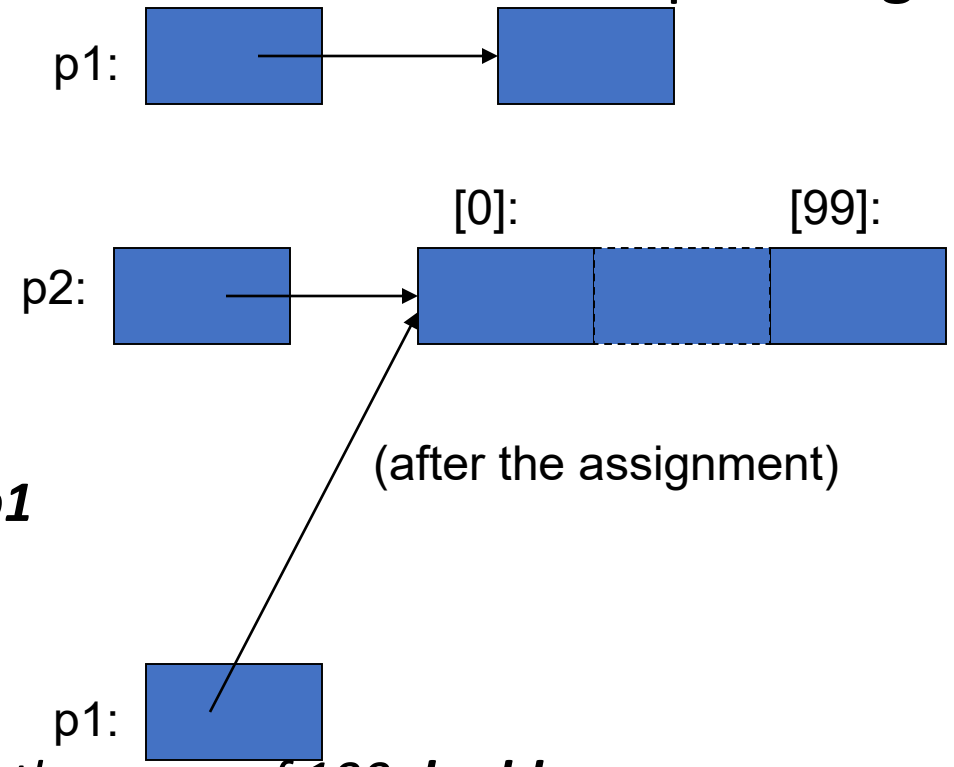




Access

- A pointer does ***not*** know the number of elements that it's pointing to

double* p1 = new double;
double* p2 = new double[100];



p1[17] = 9.4; **//** *error (obviously)*

p1 = p2; **//** *assign the value of* ***p2*** *to* ***p1***

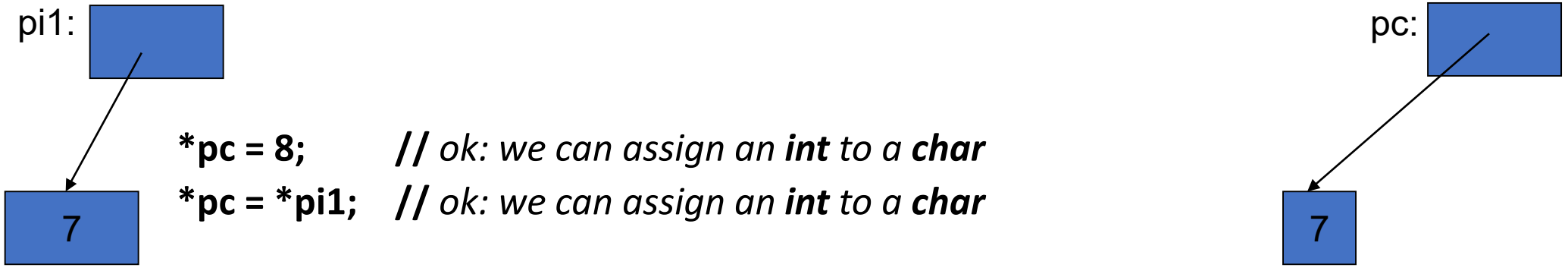
p1[17] = 9.4; **//** *now ok:* ***p1*** *now points to the array of 100* ***doubles***

Access

- A pointer ***does*** know the type of the object that it's pointing to

```
int* pi1 = new int(7);
int* pi2 = pi1;          // ok: pi2 points to the same object as pi1
double* pd = pi1;        // error: can't assign an int* to a double*
char* pc = pi1;          // error: can't assign an int* to a char*
```

- There are no implicit conversions between a pointer to one value type to a pointer to another value type
  - However, there are implicit conversions between value types




```
*pc = 8;       // ok: we can assign an int to a char
*pc = *pi1;    // ok: we can assign an int to a char
```

Pointers, arrays, and vector

- Note
  - With pointers and arrays we are “touching” hardware directly with only the most minimal help from the language. Here is where serious programming errors can most easily be made, resulting in malfunctioning programs and obscure bugs
    - Be careful and operate at this level only when you really need to
    - If you get “segmentation fault”, “bus error”, or “core dumped”, suspect an uninitialized or otherwise invalid pointer
  - vector is one way of getting almost all of the flexibility and performance of arrays with greater support from the language (read: fewer bugs and less debug time).

Vector (construction and primitive access)

```
// a very simplified vector of doubles:
class vector {
    int sz;                 // the size
    double* elem;           // a pointer to the elements
public:
    vector(int s) :sz(s), elem(new double[s]) { }       // constructor
    double get(int n) const { return elem[n]; }         // access: read
    void set(int n, double v) { elem[n]=v; }            // access: write
    int size() const { return sz; }                     // the current size
};

vector v(10);
for (int i=0; i<v.size(); ++i) { v.set(i,i); cout << v.get(i) << ' '; }
```

A problem: memory leak

```
double* calc(int result_size, int max)
{
   double* p = new double[max];          // allocate another max doubles
                                         // i.e., get max doubles from the free store
   double* result = new double[result_size];
   // … use p to calculate results to be put in result …
   return result;
}

double* r = calc(200,100);               // oops! We "forgot" to give the memory
                                          // allocated for p back to the free store
```

- Lack of de-allocation (usually called "memory leaks") can be a serious problem in real-world programs
- A program that must run for a long time can't afford any memory leaks

A problem: memory leak

```
double* calc(int result_size, int max)
{
    double* p = new double[max];        // allocate another max doubles,
                                        // i.e., get max doubles from the free store
    double* result = new double[result_size];
    // … use p to calculate results to be put in result …
    delete[ ] p;                        // de-allocate (free) that array, i.e., give the array back to the free store
    return result;
}

double* r = calc(200,100);
// use r
delete[ ] r;                            // easy to forget
```

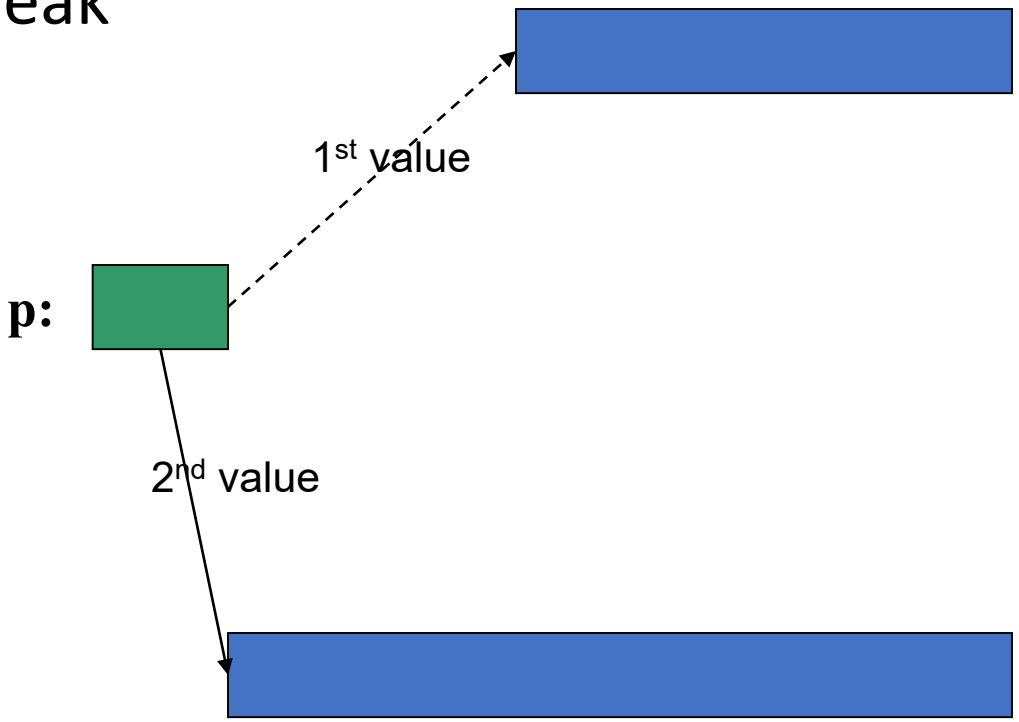
Memory leaks

- A program that needs to run “forever” can’t afford any memory leaks
  - An operating system is an example of a program that “runs forever”
- If a function leaks 8 bytes every time it is called, how many days can it run before it has leaked/lost a megabyte?
  - Trick question: not enough data to answer, but about 130,000 calls
- All memory is returned to the system at the end of the program
  - If you run using an operating system (Windows, Unix, whatever)
- Program that runs to completion with predictable memory usage may leak without causing problems
  - *i.e.,* memory leaks aren’t “good/bad” but they can be a major problem in specific circumstances

Memory leaks

- Another way to get a memory leak

```
void f()
{
   double* p = new double[27];
   // …
   p = new double[42];
   // …
   delete[] p;
}

// 1st array (of 27 doubles) leaked
```



Memory leaks

- How do we systematically and simply avoid memory leaks?
  - Don't mess directly with **new** and **delete**
    - Use **vector**, etc.
  - Or use a garbage collector
    - A garbage collector is a program the keeps track of all of your allocations and returns unused free-store allocated memory to the free store (not covered in this course; see http://www.stroustrup.com/C++.html)
    - Unfortunately, even a garbage collector doesn’t prevent all leaks
    - See also Chapter 25

A problem: memory leak

```
void f(int x)
{
    vector v(x);        // define a vector
                        // (which allocates x doubles on the free store)
    // … use v …

    // give the memory allocated by v back to the free store
    // but how? (vector's elem data member is private)
}
```

Vector (destructor)

```
// a very simplified vector of doubles:
class vector {
   int sz;                    // the size
   double* elem;              // a pointer to the elements
public:
   vector(int s)              // constructor: allocates/acquires memory
         :sz(s), elem(new double[s]) { }
   ~vector()                  // destructor: de-allocates/releases memory
         { delete[ ] elem; }
   // …
};
```

- Note: this is an example of a general and important technique:
  - acquire resources in a constructor
  - release them in the destructor
- Examples of resources: memory, files, locks, threads, sockets

A problem: memory leak

```
void f(int x)
{
   int* p = new int[x];        // allocate x ints
   vector v(x);                // define a vector (which allocates another x ints)
   // … use p and v …
   delete[ ] p;                // deallocate the array pointed to by p
   // the memory allocated by v is implicitly deleted here by vector's destructor
}
```

- The **delete** now looks verbose and ugly
  - How do we avoid forgetting to **delete[ ] p**?
  - Experience shows that we often forget
- Prefer **delete**s in destructors

Free store summary

- Allocate using **new**
  - New allocates an object on the free store, sometimes initializes it, and returns a pointer to it
    - **int* pi = new int;** // *default initialization (none for **int**)*
    - **char* pc = new char('a');** // *explicit initialization*
    - **double* pd = new double[10];** // *allocation of (uninitialized) array*
  - New throws a **bad_alloc** exception if it can't allocate (out of memory)
- Deallocate using **delete** and **delete[ ]**
  - **delete** and **delete[ ]** return the memory of an object allocated by **new** to the free store so that the free store can use it for new allocations
    - **delete pi;** // *deallocate an individual object*
    - **delete pc;** // *deallocate an individual object*
    - **delete[ ] pd;** // *deallocate an array*
  - Delete of a zero-valued pointer ("the null pointer") does nothing
    - **char* p = 0;** // *C++11 would say* **char* p = nullptr;**
    - **delete p;** // *harmless*

Vector: Initialization, Copy and Move

Initialization: initialize lists

- We would like simple, general, and flexible initialization
  - So we provide suitable constructors, including

```
class vector  {
   // …
public:
    vector(int s);                                  // constructor (s is the element count)

    vector(std::initializer_list<double> lst);      // initializer-list constructor
    // …
 };

vector v1(20);                  // 20 elements, each initialized to 0
vector v2 {1,2,3,4,5};          // 5 elements: 1,2,3,4,5
```

Initialization: initializer lists

- We would like simple, general, and flexible initialization
  - So we provide suitable constructors

```
vector::vector(int s)    // constructor (s is the element count)
    :sz{s}, elem{new double[s]} { }
{
    for (int i=0; i<sz; ++i) elem[i]=0;
}

vector::vector(std::initializer_list<double> lst)   // initializer-list constructor
    :sz{lst.size()}, elem{new double[sz]} { }
{
    std::copy(lst.begin(),lst.end(),elem); // copy lst to elem
}
vector v1(20);              // 20 elements, each initialized to 0
vector v2 {1,2,3,4,5};  // 5 elements: 1,2,3,4,5
```

Initialization: lists and sizes

- If we initialize a vector by 17 is it
  - 17 elements (with value 0)?
  - 1 element with value 17?
- By convention use
  - () for number of elements
  - {} for elements
- For example
  - **vector v1(17);** **//** *17 elements, each with the value 0*
  - **vector v2 {17};** **//** *1 element with value 17*

A problem

- Copy doesn't work as we would have hoped (expected?)

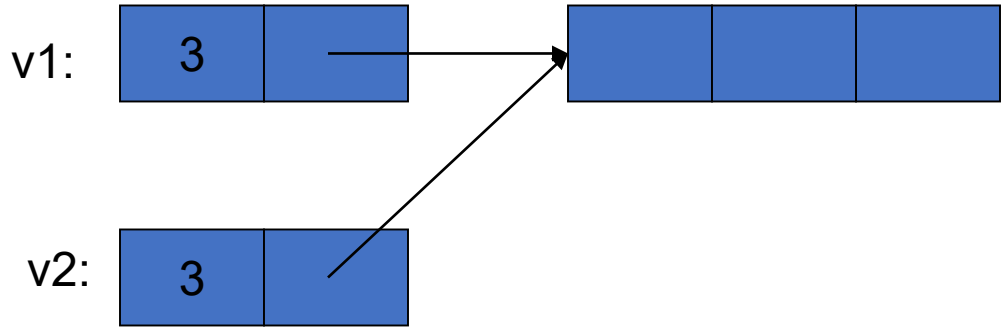
```
void f(int n)
{
    vector v(n);            // define a vector
    vector v2 = v;          // what happens here?
                            // what would we like to happen?
    vector v3;
    v3 = v;                 // what happens here?
                            // what would we like to happen?
}
```

- Ideally: **v2** and **v3** become copies of **v** (that is, **=** makes copies**)**
  - And all memory is returned to the free store upon exit from **f()**
- That's what the standard **vector** does,
  - but it's not what happens for our still-too-simple **vector**

Naïve copy initialization (the default)

- By default “copy” means “copy the data members”

```
void f(int n)
{
   vector v1(n);
   vector v2 = v1; // initialization:
                   // by default, a copy of a class copies its members
                   // so sz and elem are copied
}
```

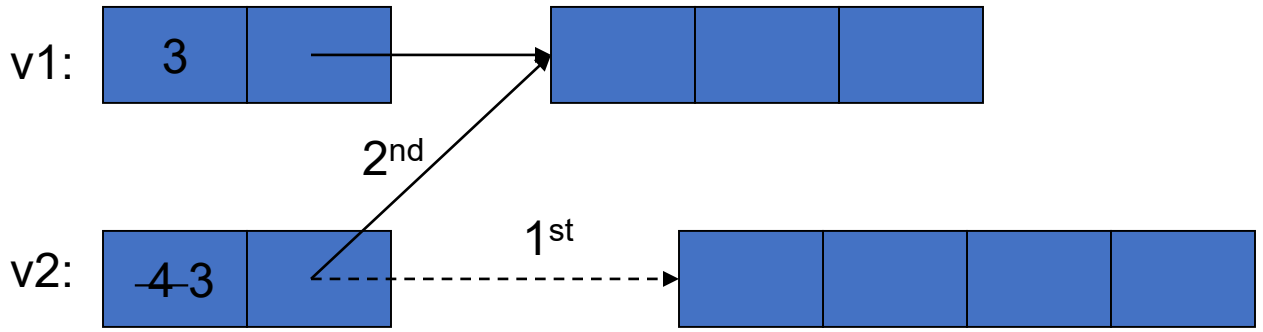


Disaster when we leave f()!

v1’s elements are deleted twice (by the 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
}
```



Disaster when we leave f()!

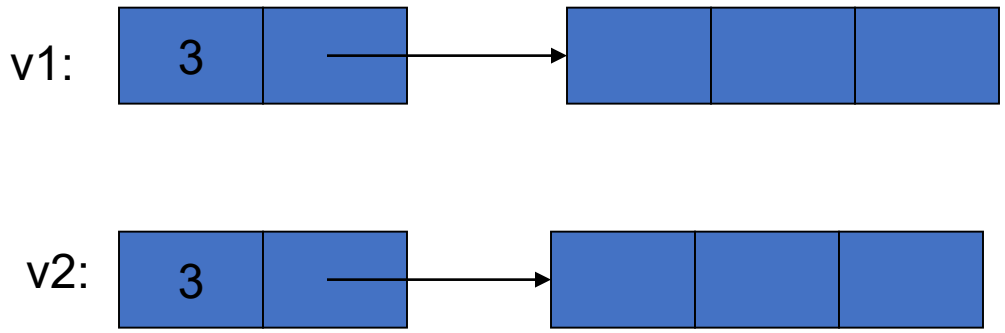
- v1's elements are deleted twice (by the destructor)
- memory leak: v2's elements are not deleted

Copy constructor (initialization)

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

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

Copy with copy constructor

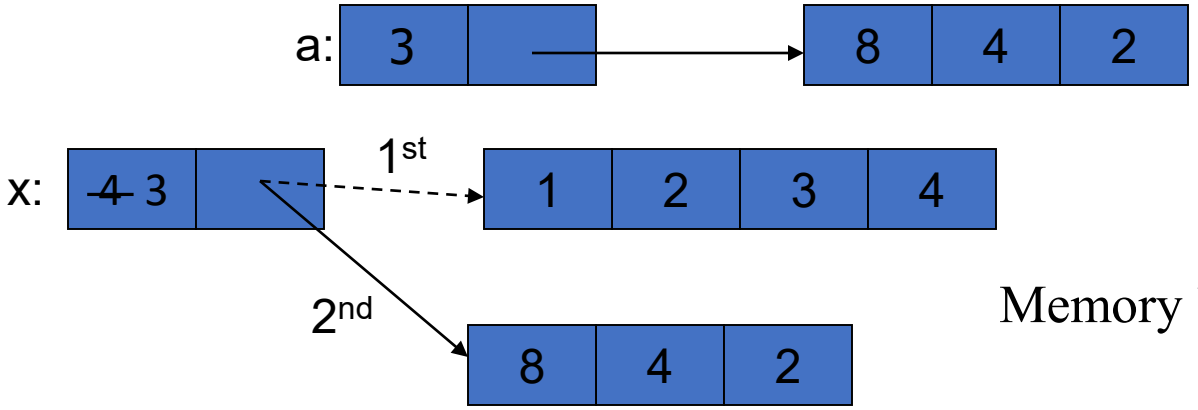
```
void f(int n)
{
   vector v1(n);
   vector v2 = v1;          // copy using the copy constructor
                            // the for loop copies each value from v1 into v2
}
```



The destructor correctly deletes all elements
(once only for each vector)

Copy assignment

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




```
x=a;
```

Memory leak?

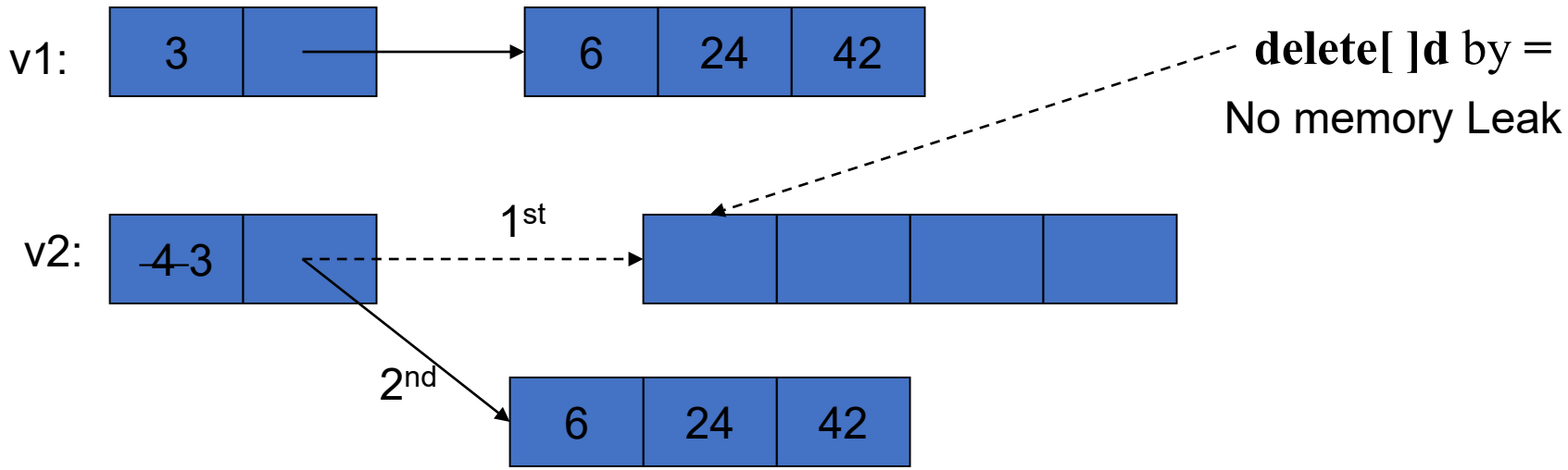
Operator = must copy a's elements

Copy assignment

```
vector& vector::operator=(const vector& a)
   // like copy constructor, but we must deal with old elements
   // make a copy of a then replace the current sz and elem with a's
{
   double* p = new double[a.sz];                  // allocate new space
   for (int i = 0; i<a.sz; ++i) p[i] = a.elem[i]; // copy elements
   delete[ ] elem;                                // deallocate old space
   sz = a.sz;                                     // set new size
   elem = p;                                      // set new elements
   return *this;                                  //  return a self-reference
                                                  // The this pointer is explained in 17.10
}
```

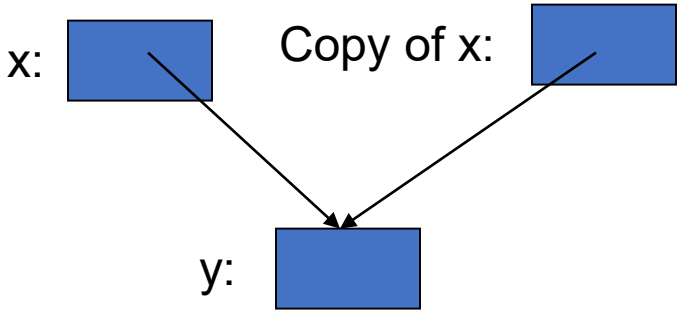
Copy with copy assignment

```
void f(int n)
{
   vector v1 {6,24,42};
   vector v2(4);
   v2 = v1;                    // assignment
}
```

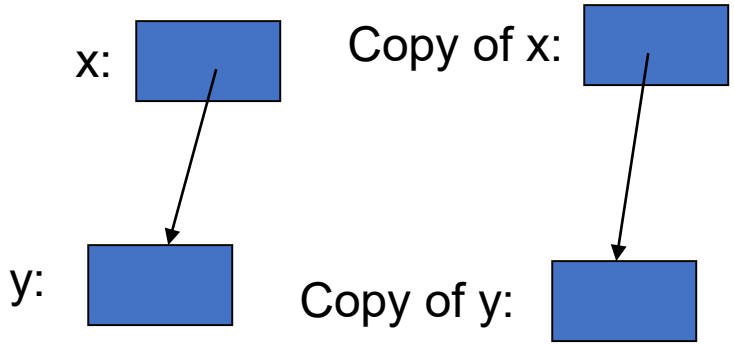


Copy terminology

- Shallow copy: copy only a pointer so that the two pointers now refer to the same object
  - What pointers and references do
- Deep copy: copy what the pointer points to so that the two pointers now each refer to a distinct object
  - What **vector**, **string**, etc. do
  - Requires copy constructors and copy assignments for container classes
  - Must copy “all the way down” if there are more levels in the object

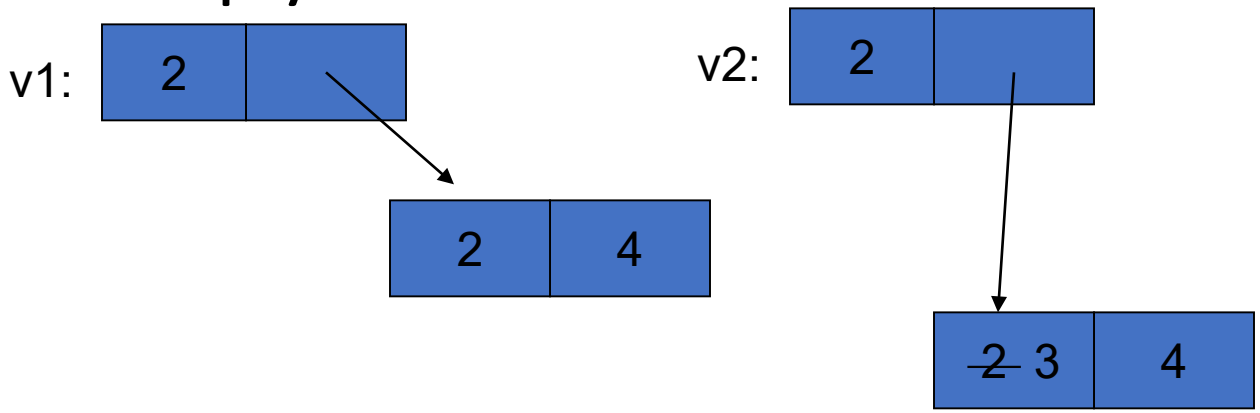


Shallow copy

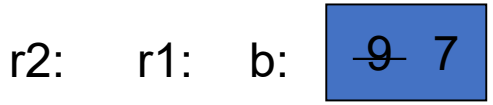


Deep copy

Deep and shallow copy




```
vector<int> v1 {2,4};
vector<int> v2 = v1;        // deep copy (v2 gets its own copy of v1's elements)
v2[0] = 3;                  // v1[0] is still 2
```




```
int b = 9;
int& r1 = b;
int& r2 = r1;       // shallow copy (r2 refers to the same variable as r1)
r2 = 7;             // b becomes 7
```

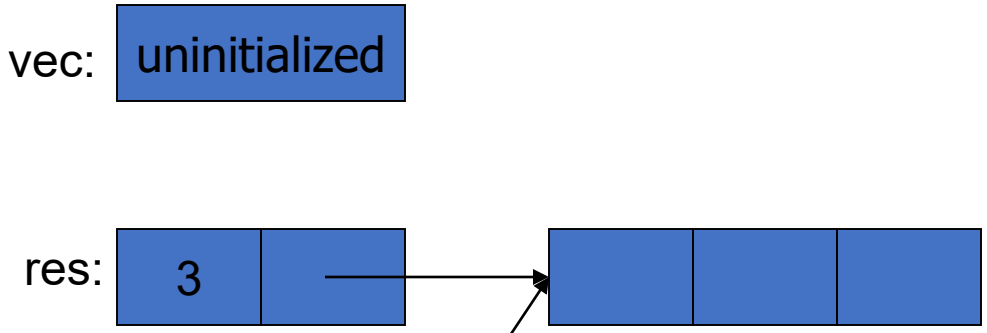
Move

- Consider

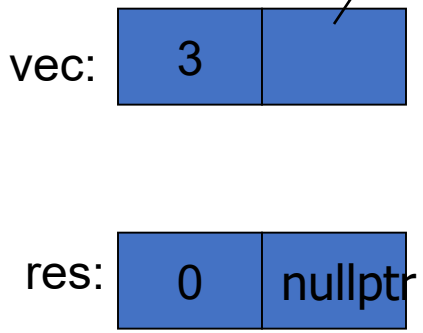
```
vector fill(istream& is)
{
    vector res;
    for (double x; is>>x; ) res.push_back(x);
    return res;          // returning a copy of res could be expensive
                         // returning a copy of res would be silly!
}

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

What we want: Move

- Before **return res;** in **fill()**

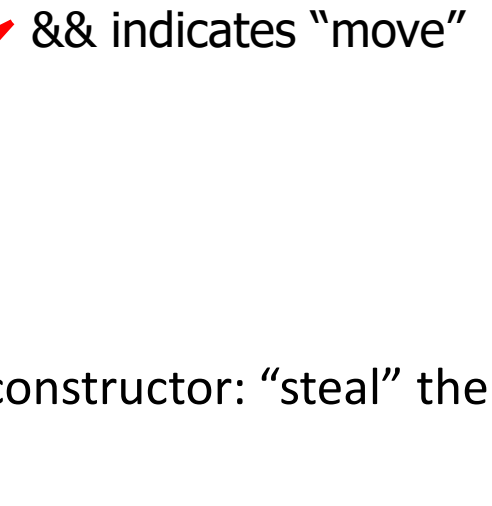


- After **return res;** (after **vector vec = fill(cin);**)



Move Constructor and assignment

- Define move operations to “steal” representation




```
class vector {
     int sz;
     double* elem;
public:
    vector(vector&&);                  // move constructor: “steal” the elements

    vector& operator=(vector&&);       // move assignment:
                                       // destroy target and “steal” the elements

    // . . .
};
```

Move implementation

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

Move implementation

```
vector& vector::operator=(vector&& a)       // move assignment
{
    delete[] elem;                          // deallocate old space
    elem = a.elem;                          // copy a's elem and sz
    sz = a.sz;
    a.elem = nullptr;                       // make a the empty vector
    a.sz = 0;
    return *this;                           // return a self-reference
}
```

Essential operations

- Constructors from one or more arguments
- Default constructor

- Copy constructor (copy object of same type)
- Copy assignment (copy object of same type)
- Move constructor (move object of same type)
- Move assignment (move object of same type)
- Destructor

- If you define one of the last 5, define them all

Vector: Changing Size

Changing vector size

- Fundamental problem addressed
  - We (humans) want abstractions that can change size (*e.g.*, a vector where we can change the number of elements). However, in computer memory everything must have a fixed size, so how do we create the illusion of change?
- Given

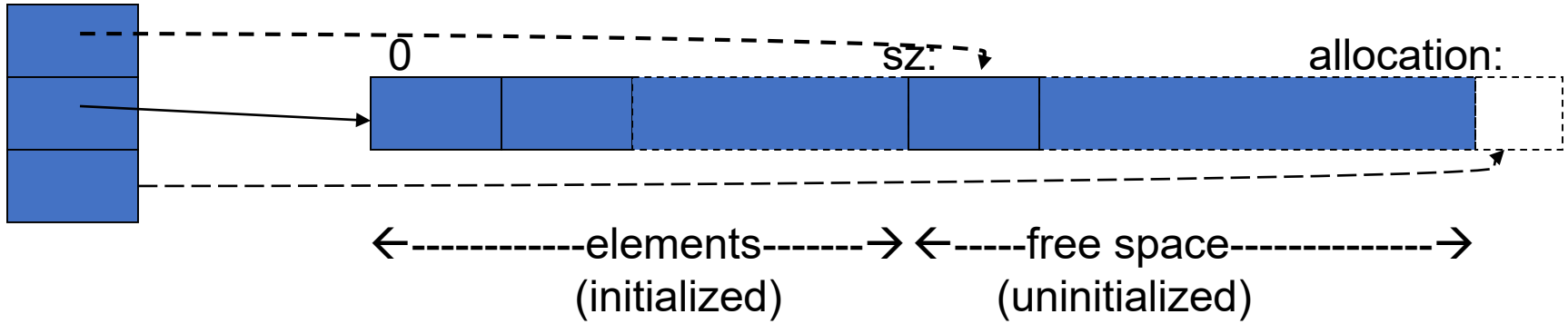
  vector v(n); **// *v.size()==n***

  we can change its size in three ways
  - Resize it
    - **v.resize(10);** **// *v* now has *10* elements**
  - Add an element
    - **v.push_back(7);** **// add an element with the value *7* to the end of *v***
      // *v.size()* increases by *1*
  - Assign to it
    - **v = v2;** **// *v* is now a copy of *v2***
      // *v.size()* now equals *v2.size()*

Representing vector

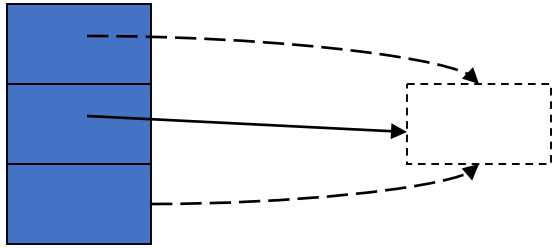
- If you **resize()** or **push_back()** once, you'll probably do it again;
  - Let's prepare for that by sometimes keeping a bit of free space for future expansion

```
class vector {
    int sz;
    double* elem;
    int space;      // number of elements plus "free space"
                    // (the number of "slots" for new elements)
public:
    // …
};
```

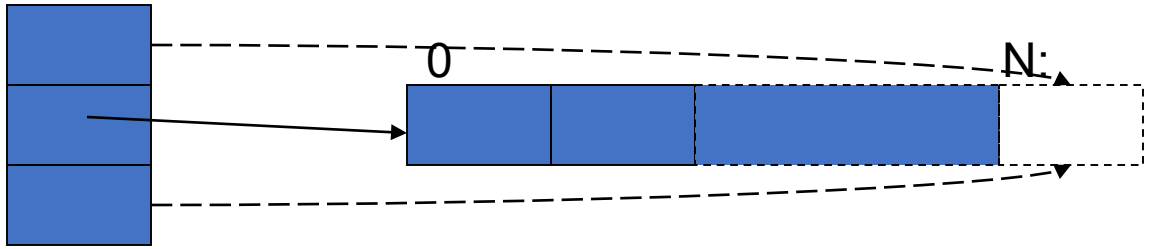


Representing vector

- An empty vector (no free store use):



- A vector(n) (no free space):



vector::reserve()

- First deal with space (allocation); given space all else is easy
  - Note: **reserve()** doesn't mess with size or element values

```
void vector::reserve(int newalloc)
   // make the vector have space for newalloc elements
{
   if (newalloc<=space) return;              // never decrease allocation
   double* p = new double[newalloc];         // allocate new space
   for (int i=0; i<sz; ++i) p[i]=elem[i];    // copy old elements
   delete[ ] elem;                           // deallocate old space
   elem = p;
   space = newalloc;
}
```

vector::resize()

- Given **reserve()**, **resize()** is easy
  - **reserve()** deals with space/allocation
  - **resize()** deals with element values

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

vector::push_back()

- Given **reserve()**, **push_back()** is easy
  - **reserve()** deals with space/allocation
  - **push_back()** just adds a value

```
void vector::push_back(double d)
   // increase vector size by one
   // initialize the new element with d
{
   if (sz==0)                    // no space: grab some
         reserve(8);
   else if (sz==space)           // no more free space: get more space
         reserve(2*space);
   elem[sz] = d;                 // add d at end
   ++sz;                         // and increase the size (sz is the number of elements)
}
```

resize() and push_back()

```
class vector {                  // an almost real vector of doubles
   int sz;                      // the size
   double* elem;                // a pointer to the elements
   int space;                   // size+free_space
public:
   // … constructors and destructors …

   double& operator[ ](int n) { return elem[n]; }      // access: return reference
   int size() const { return sz; }                     // current size

   void resize(int newsize);                           // grow
   void push_back(double d);                           // add element

   void reserve(int newalloc);                         // get more space
   int capacity() const { return space; }              // current available space
};
```

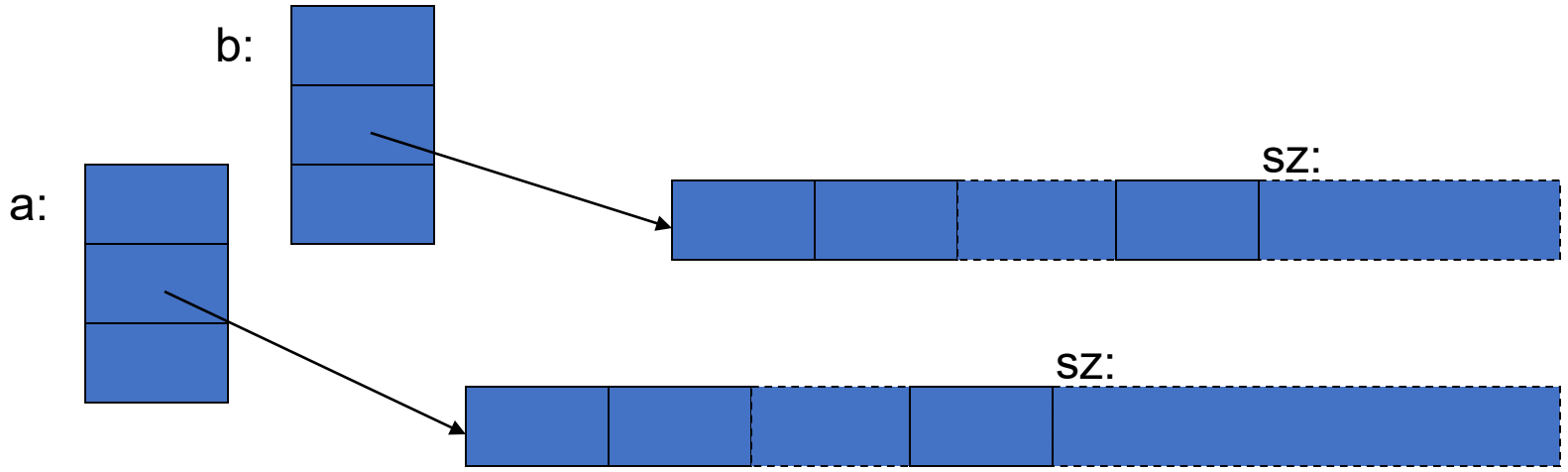
Assignment

- Copy and swap is a powerful general idea

```
vector& vector::operator=(const vector& a)
   // like copy constructor, but we must deal with old elements
   // make a copy of a then replace the current sz and elem with a's
{
   double* p = new double[a.sz];                  // allocate new space
   for (int i = 0; i<a.sz; ++i) p[i] = a.elem[i]; // copy elements
   delete[ ] elem;                                // deallocate old space
   sz = a.sz;                                     // set new size
   elem = p;                                      // set new elements
   return *this;                                  //  return a self-reference
}
```

Optimize assignment

- “Copy and swap” is the most general idea
  - but not always the most efficient
  - What if there already is sufficient space in the target vector?
    - Then just copy!
    - For example: **a = b;**



Optimized assignment

```
vector& vector::operator=(const vector& a)
{
    if (this==&a) return *this;  // self-assignment, no work needed

    if (a.sz<=space) {              // enough space, no need for new allocation
            for (int i = 0; i<a.sz; ++i) elem[i] = a.elem[i];       // copy elements
            sz = a.sz;
            return *this;
    }
    double* p = new double[a.sz];                                   // copy and swap
    for (int i = 0; i<a.sz; ++i) p[i] = a.elem[i];
    delete[ ] elem;
    sz = a.sz;
    space = a.sz;
    elem = p;
    return *this;
}
```

Templates and Range Checking

Templates

- But we don’t just want vector of double
- We want vectors with element types we specify
  - **vector<double>**
  - **vector<int>**
  - **vector<Month>**
  - **vector<Record*>** // *vector of pointers*
  - **vector<vector<Record>>** // *vector of vectors*
  - **vector<char>**
- We must make the element type a parameter to **vector**
- **vector** must be able to take both built-in types and user-defined types as element types
- This is not some magic reserved for the compiler; we can define our own parameterized types, called “templates”

Templates

- The basis for generic programming in C++
  - Sometimes called “parametric polymorphism”
    - Parameterization of types (and functions) by types (and integers)
  - Unsurpassed flexibility and performance
    - Used where performance is essential (*e.g.,* hard real time and numerics)
    - Used where flexibility is essential (*e.g.,* the C++ standard library)

- Template definitions

```
template<class T, int N> class Buffer { /* … */ };
template<class T, int N> void fill(Buffer<T,N>& b) { /* … */  }
```

- Template specializations (instantiations)

```
// for a class template, you specify the template arguments:
Buffer<char,1024> buf;          // for buf, T is char and N is 1024

// for a function template, the compiler deduces the template arguments:
fill(buf);                      // for fill(), T is char and N is 1024; that's what buf has
```

Parameterize with element type

```
// an almost real vector of Ts:
template<class T> class vector {
   // …
};
vector<double> vd;                  // T is double
vector<int> vi;                     // T is int
vector<vector<int>> vvi;            // T is vector<int>
                                    // in which T is int
vector<char> vc;                    // T is char
vector<double*> vpd;                // T is double*
vector<vector<double>*> vvpd;       // T is vector<double>*
                                    // in which T is double
```

Basically, **vector<double>** is

```
// an almost real vector of doubles:
class vector {
    int sz;                     // the size
    double* elem;               // a pointer to the elements
    int space;                  // size+free_space
public:
    vector() : sz(0), elem(0), space(0) { }          // default constructor
    vector(const vector&);                           // copy constructor
    vector& operator=(const vector&);                // copy assignment
    ~vector() { delete[ ] elem; }                    // destructor

    double& operator[ ] (int n) { return elem[n]; }  // access: return reference
    int size() const { return sz; }                  // the current size

    // …
};
```

Basically, **vector<char>** is

```
// an almost real vector of chars:
class vector {
    int sz;                     // the size
    char* elem;                 // a pointer to the elements
    int space;                  // size+free_space
public:
    vector() : sz{0}, elem{0}, space{0} { }     // default constructor
    vector(const vector&);                      // copy constructor
    vector& operator=(const vector&);           // copy assignment
    ~vector() { delete[ ] elem; }               // destructor

    char& operator[ ] (int n) { return elem[n]; }   // access: return reference
    int size() const { return sz; }                 // the current size

    // …
};
```

Basically, **vector<T>** is

```
// an almost real vector of Ts:
template<class T> class vector {      // read "for all types T" (just like in math)
   int sz;                            // the size
   T* elem;                           // a pointer to the elements
   int space;                         // size+free_space
public:
   vector() : sz{0}, elem{0}, space{0};          // default constructor
   vector(const vector&);                        // copy constructor
   vector& operator=(const vector&);             // copy assignment
   vector(const vector&&);                       // move constructor
   vector& operator=(vector&&);                  // move assignment

   ~vector() { delete[ ] elem; }                 // destructor
   // …
};
```

Basically, **vector<T>** is

```
// an almost real vector of Ts:
template<class T> class vector {      // read "for all types T" (just like in math)
   int sz;                            // the size
   T* elem;                           // a pointer to the elements
   int space;                         // size+free_space
public:
   // … constructors and destructors …

   T& operator[ ] (int n) { return elem[n]; }      // access: return reference
   int size() const { return sz; }                 // the current size

   void resize(int newsize);                       // grow
   void push_back(double d);                       // add element

   void reserve(int newalloc);                     // get more space
   int capacity() const { return space; }          // current available space
   // …
};
```

Templates

- Problems ("there is no free lunch")
  - Poor error diagnostics
    - Often spectacularly poor (but getting better in C++11; much better in C++14)
  - Delayed error messages
    - Often at link time
  - All templates must be fully defined in each translation unit
    - So place template definitions in header files
- Recommendation
  - Use template-based libraries
    - Such as the C++ standard library
      - *E.g.,* **vector**, **sort()**
      - Soon to be described in some detail
  - Initially, write only very simple templates yourself
    - Until you get more experience

Range checking

```
// an almost real vector of Ts:

struct out_of_range { /* … */ };

template<class T> class vector {
   // …
   T& operator[ ](int n);                  // access
   // …
};

template<class T>  T& vector<T>::operator[ ](int n)
{
   if (n<0 || sz<=n) throw out_of_range();
   return elem[n];
}
```

Range checking

```
void fill_vec(vector<int>& v, int n)          // initialize v with factorials
{
   for (int i=0; i<n; ++i) v.push_back(factorial(i));
}

int main()
{
   vector<int> v;
   try {
            fill_vec(v,10);
            for (int i=0; i<=v.size(); ++i)
                     cout << "v[" << i << "]==" << v[i] << '\n';
   }
   catch (out_of_range) {                     // we'll get here (why?)
            cout << "out of range error";
            return 1;
   }
}
```

Exception handling

- We use exceptions to report errors
- We must ensure that use of exceptions
  - Doesn’t introduce new sources of errors
  - Doesn’t complicate our code
  - Doesn’t lead to resource leaks

STL
(The containers, iterators and algorithms)

- STL – the containers and algorithms part of the C++ standard library

Common tasks

- Collect data into containers
- Organize data
  - For printing
  - For fast access
- Retrieve data items
  - By index (e.g., get the **N**th element)
  - By value (e.g., get the first element with the value **“Chocolate”**)
  - By properties (e.g., get the first elements where “**age<64”**)
- Add data
- Remove data
- Sorting and searching
- Simple numeric operations

Observation

We can (already) write programs that are very similar independent of the data type used

- Using an **int** isn't that different from using a **double**
- Using a **vector<int>** isn't that different from using a **vector<string>**

Ideals

We'd like to write common programming tasks so that we don't have to re-do the work each time we find a new way of storing the data or a slightly different way of interpreting the data

- Finding a value in a **vector** isn't all that different from finding a value in a **list** or an array
- Looking for a **string** ignoring case isn't all that different from looking at a **string** not ignoring case
- Graphing experimental data with exact values isn't all that different from graphing data with rounded values
- Copying a file isn't all that different from copying a vector

Ideals (continued)

- Code that's
  - Easy to read
  - Easy to modify
  - Regular
  - Short
  - Fast
- Uniform access to data
  - Independently of how it is stored
  - Independently of its type
- …

Ideals (continued)

- …
- Type-safe access to data
- Easy traversal of data
- Compact storage of data
- Fast
  - Retrieval of data
  - Addition of data
  - Deletion of data
- Standard versions of the most common algorithms
  - Copy, find, search, sort, sum, …

Examples

- Sort a vector of strings
- Find an number in a phone book, given a name
- Find the highest temperature
- Find all values larger than 800
- Find the first occurrence of the value 17
- Sort the telemetry records by unit number
- Sort the telemetry records by time stamp
- Find the first value larger than “Petersen”?
- What is the largest amount seen?
- Find the first difference between two sequences
- Compute the pairwise product of the elements of two sequences
- What are the highest temperatures for each day in a month?
- What are the top 10 best-sellers?
- What’s the entry for “C++” (say, in Google)?
- What’s the sum of the elements?

The STL

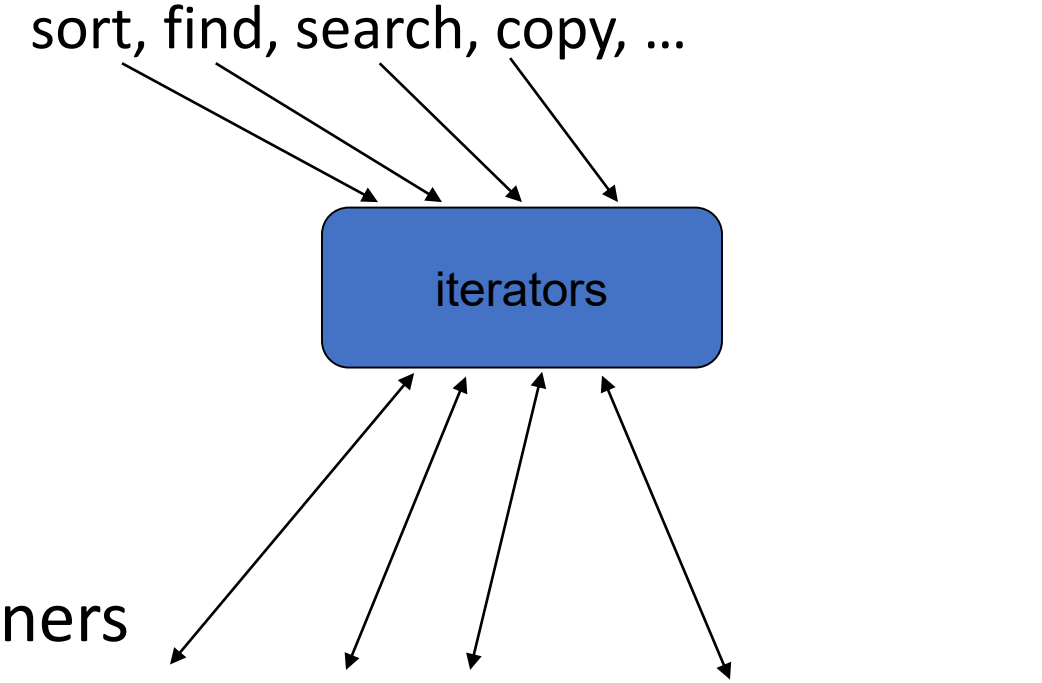
- Part of the ISO C++ Standard Library
- Mostly non-numerical
  - Only 4 standard algorithms specifically do computation
    - Accumulate, inner_product, partial_sum, adjacent_difference
  - Handles textual data as well as numeric data
    - E.g. string
  - Deals with organization of code and data
    - Built-in types, user-defined types, and data structures
- Optimizing disk access was among its original uses
  - Performance was always a key concern

The STL

- Designed by Alex Stepanov
- General aim: The most general, most efficient, most flexible representation of concepts (ideas, algorithms)
  - Represent separate concepts separately in code
  - Combine concepts freely wherever meaningful
- General aim to make programming “like math”
  - or even “Good programming *is* math”
  - works for integers, for floating-point numbers, for polynomials, for …

Basic model

- Algorithms



- Containers

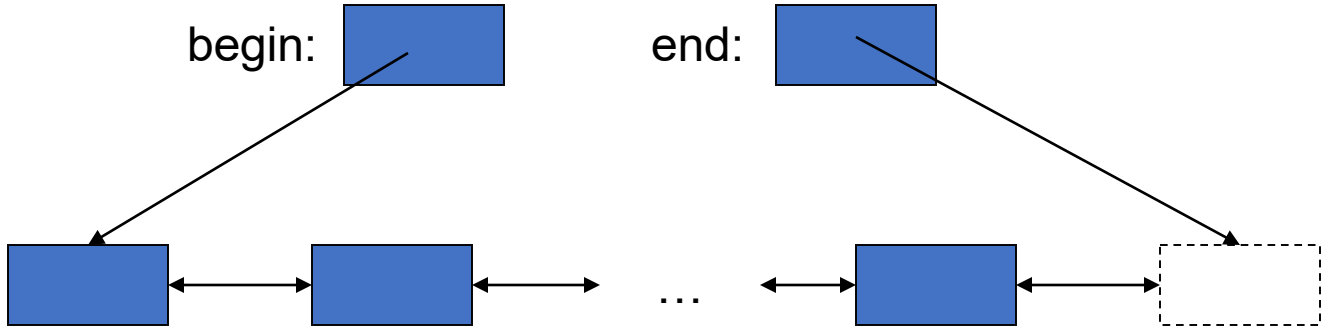
- Separation of concerns
  - Algorithms manipulate data, but don’t know about containers
  - Containers store data, but don’t know about algorithms
  - Algorithms and containers interact through iterators
- Each container has its own iterator types

The STL

- An ISO C++ standard framework of about 10 containers and about 60 algorithms connected by iterators
  - Other organizations provide more containers and algorithms in the style of the STL
    - Boost.org, Microsoft, SGI, …
- Probably the currently best known and most widely used example of generic programming

Basic model

- A pair of iterators defines a sequence
  - The beginning (points to the first element – if any)
  - The end (points to the one-beyond-the-last element)

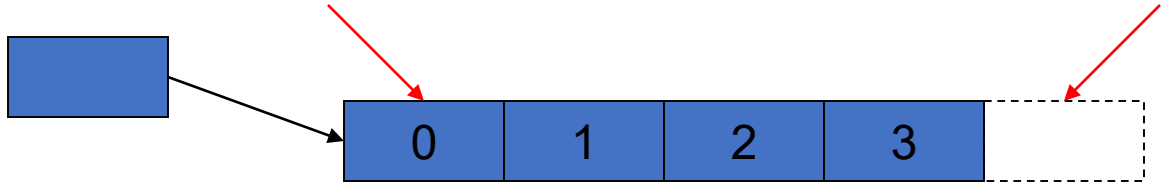


- An iterator is a type that supports the "iterator operations"
  - ++ Go to next element
  - * Get value
  - == Does this iterator point to the same element as that iterator?
- Some iterators support more operations (e.g. --, +, and [ ])

Containers

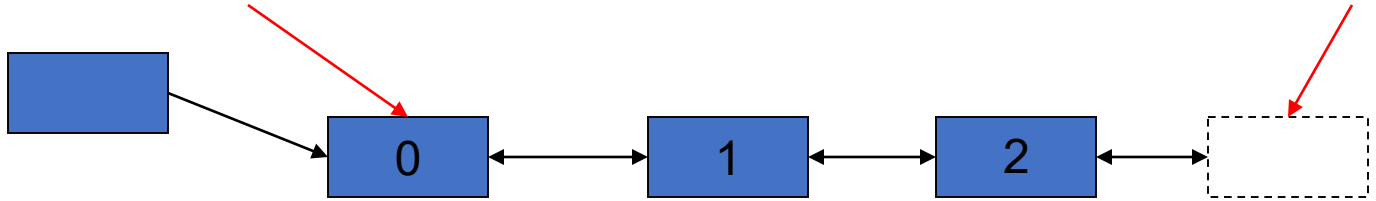
(hold sequences in difference ways)

- **vector**



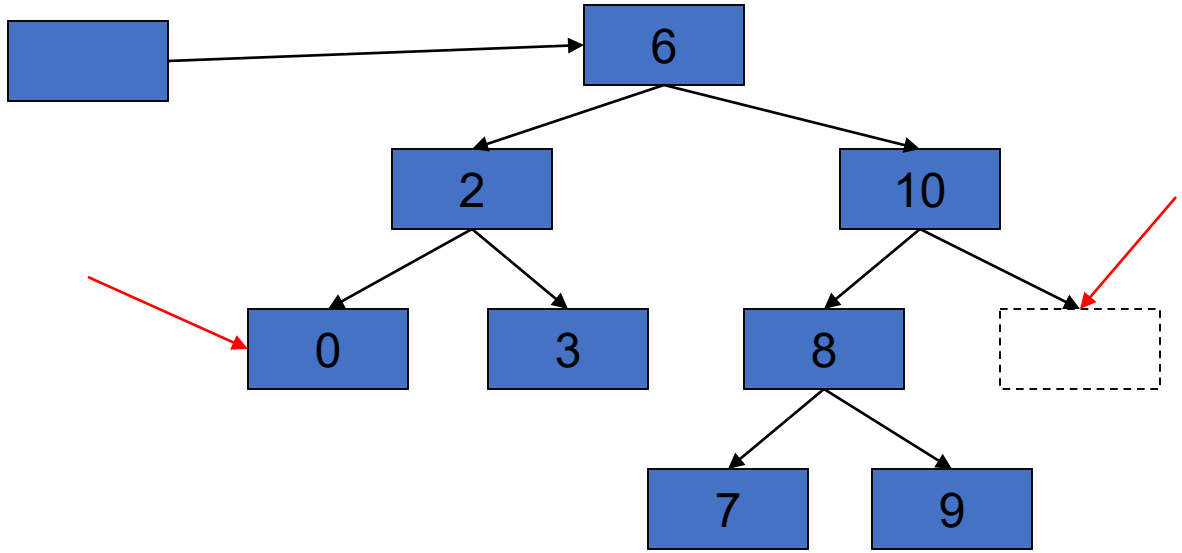
- **list**

(doubly linked)

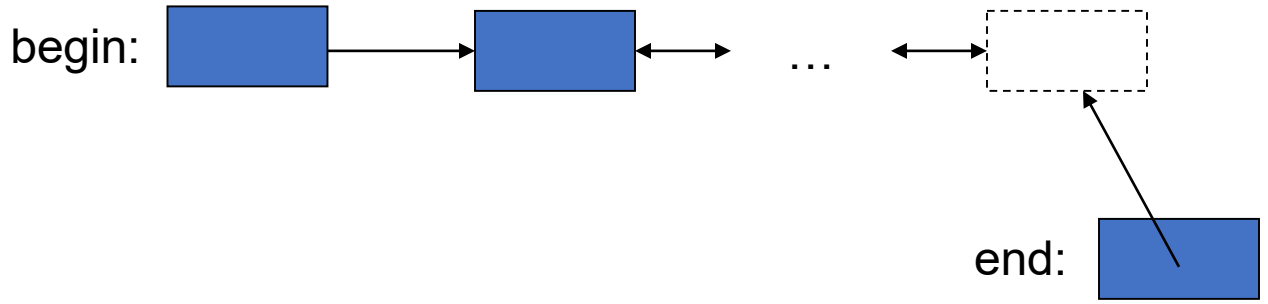


- **set**

(a kind of tree)



The simplest algorithm: **find()**




```
// Find the first element that equals a value

template<class In, class T>
In find(In first, In last, const T& val)
{
   while (first!=last && *first != val) ++first;
   return first;
}

void f(vector<int>& v, int x)          // find an int in a vector
{
   vector<int>::iterator p = find(v.begin(),v.end(),x);
   if (p!=v.end()) { /* we found  x */ }
   // …
}
```

We can ignore (“abstract away”) the differences between containers

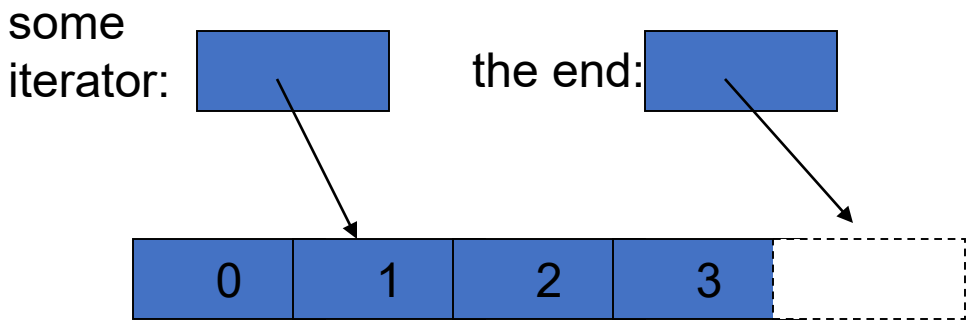
find()

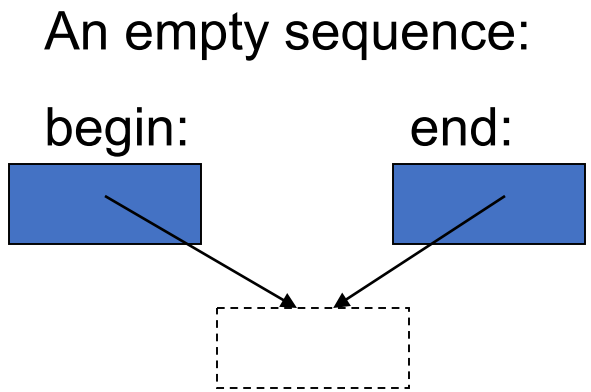
generic for both element type and container type

```
void f(vector<int>& v, int x)                // works for vector of ints
{
    vector<int>::iterator p = find(v.begin(),v.end(),x);
    if (p!=v.end()) { /* we found  x */ }
    // …
}

void f(list<string>& v, string x)            // works for list of strings
{
    list<string>::iterator p = find(v.begin(),v.end(),x);
    if (p!=v.end()) { /* we found  x */ }
    // …
}

void f(set<double>& v, double x)             // works for set of doubles
{
    set<double>::iterator p = find(v.begin(),v.end(),x);
    if (p!=v.end()) { /* we found  x */ }
    // …
}
```

Algorithms and iterators

- An iterator points to (refers to, denotes) an element of a sequence
- The end of the sequence is “one past the last element”
  - ***not*** “the last element”
  - That’s necessary to elegantly represent an empty sequence
  - One-past-the-last-element isn’t an element
    - You can compare an iterator pointing to it
    - You can’t dereference it (read its value)
- Returning the end of the sequence is the standard idiom for “not found” or “unsuccessful”

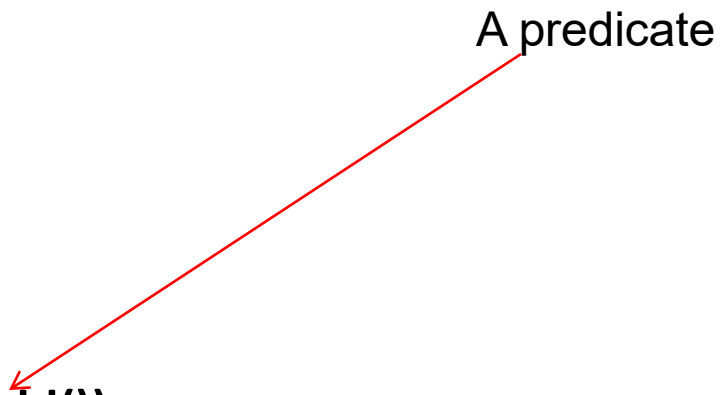




Simple algorithm: **find_if()**

- Find the first element that matches a criterion (predicate)
  - Here, a predicate takes one argument and returns a **bool**

```
template<class In, class Pred>
In find_if(In first, In last, Pred pred)
{
    while (first!=last && !pred(*first)) ++first;
    return first;
}

void f(vector<int>& v)
{
    vector<int>::iterator p = find_if(v.begin(),v.end,Odd());
    if (p!=v.end()) { /* we found an odd number */ }
    // …
}
```



Predicates

- A predicate (of one argument) is a function or a function object that takes an argument and returns a **bool**
- For example
  - A function

```
bool odd(int i) { return i%2; }        // % is the remainder (modulo) operator
odd(7);                                // call odd: is 7 odd?
```

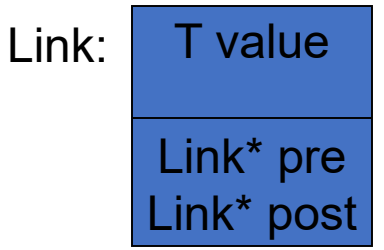
  - A function object

```
struct Odd {
        bool operator()(int i) const { return i%2; }
};
Odd odd;                               // make an object odd of type Odd
odd(7);                                // call odd: is 7 odd?
```

vector

```
template<class T> class vector {
    T* elements;
    // …
    using value_type = T;
    using iterator = ???;                  // the type of an iterator is implementation defined
                                           // and it (usefully) varies (e.g. range checked iterators)
                                           // a vector iterator could be a pointer to an element
    using const_iterator = ???;

    iterator begin();                      // points to first element
    const_iterator begin() const;
    iterator end();                        // points to one beyond the last element
    const_iterator end() const;

    iterator erase(iterator p);                    // remove element pointed to by p
    iterator insert(iterator p, const T& v);       // insert a new element v before p
};
```

list




```
template<class T> class list {
    Link* elements;
    // …
    using value_type = T;
    using iterator = ???;                   // the type of an iterator is implementation defined
                                            // and it (usefully) varies (e.g. range checked iterators)
                                            // a list iterator could be a pointer to a link node
    using const_iterator = ???;

    iterator begin();                       // points to first element
    const_iterator begin() const;
    iterator end();                         // points to one beyond the last element
    const_iterator end() const;

    iterator erase(iterator p);                     // remove element pointed to by p
    iterator insert(iterator p, const T& v);        // insert a new element v before p
};
```

Vector vs. List

- By default, use a **vector**
  - You need a reason not to
  - You can “grow” a vector (e.g., using **push_back()**)
  - You can **insert()** and **erase()** in a vector
  - Vector elements are compactly stored and contiguous
  - For small vectors of small elements all operations are fast
    - compared to lists
- If you don’t want elements to move, use a **list**
  - You can “grow” a list (e.g., using **push_back()** and **push_front()**)
  - You can **insert()** and **erase()** in a list
  - List elements are separately allocated
- Note that there are more containers, e.g.,
  - map
  - unordered_map

Some useful standard headers

- **<iostream>** I/O streams, cout, cin, …
- **<fstream>** file streams
- **<algorithm>** sort, copy, …
- **<numeric>** accumulate, inner_product, …
- **<functional>** function objects
- **<string>**
- **<vector>**
- **<map>**
- **<unordered_map>** hash table
- **<list>**
- **<set>**