DM560

Introduction to Programming in C++

Vector and Free Store (Vectors and Arrays)

Marco Chiarandini

Department of Mathematics & Computer Science
University of Southern Denmark

[*Based on slides by Bjarne Stroustrup*]

Outline

1. Initialization
2. Copy
3. Move
4. Arrays

Overview

- Vector revisited: How are they implemented?
- Pointers and free store
- Destructors
- Initialization
- Copy and move
- Arrays
- Array and pointer problems
- Changing size
- Templates
- Range checking and exceptions

Reminder

Why look at the `vector` implementation?

- To see how the standard library `vector` really works
- To introduce basic concepts and language features
  - ✔ Free store (heap)
  - Copy and move
  - Dynamically growing data structures
- 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 so far):

```
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
};
```

Outline

1. Initialization
2. Copy
3. Move
4. Arrays

Initialization: Initializer Lists

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

```
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
```

```
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
}
```

Initialization

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
```

Initialization: Explicit Constructors

A problem:

- A constructor taking a single argument defines a conversion from the argument type to the constructor's type
- Our vector had `vector::vector(int)`, so

```
vector v1 = 7;       // v1 has 7 elements, each with the value 0

void do_something(vector v)
do_something(7);    // call do_something() with a vector of 7 elements
```

This is very error-prone.

- Unless, of course, that's what we wanted
- For example

```
complex<double> d = 2.3; // convert from double to complex<double>
```

Initialization: Explicit Constructors

A solution:
Declare constructors taking a single argument explicit unless you want a conversion from the argument type to the constructor's type

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

```
vector v1 = 7;      // error: no implicit conversion from int

void do_something(vector v);
do_something(7);   // error: no implicit conversion from int
```

Outline

1. Initialization

2. Copy

3. Move

4. Arrays

A Problem

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

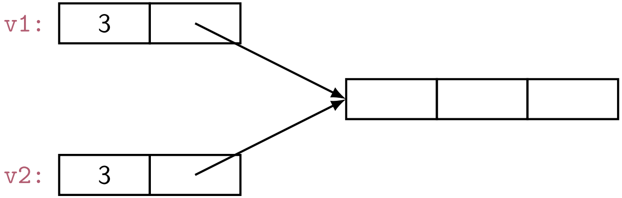
```
void f(int n)
{
    vector v(n);            // define a vector
    vector v2 = v;          // what happens here?
                            // what would we like to happen?
    vector v3;
    v3 = v;                 // what happens here?
                            // what would we like to happen?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

- 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

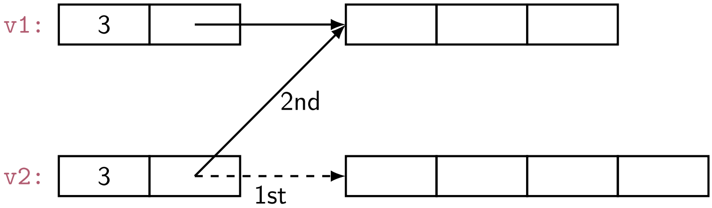
```
void f(int n)
{
    vector v1(n);
    vector v2 = v1;      // initialization:
                          // by default, a copy of a class copies its members
                          // so sz and elem are copied
}
```



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

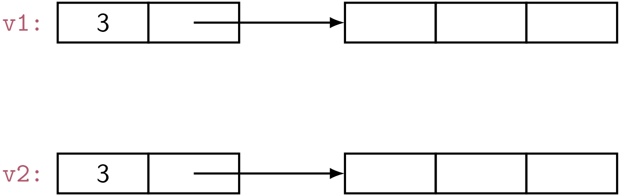
Copy Constructor (Initialization)

```
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
}
```

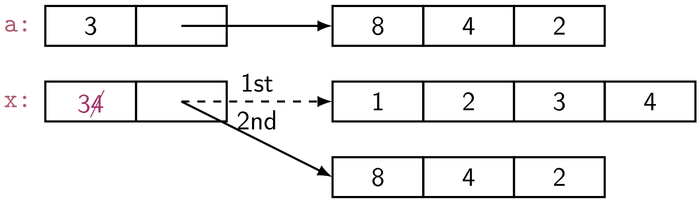


The destructor correctly deletes all elements
(once only for each vector)

Copy Assignment

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

```
x=a;
```



Memory leak? (no)

Operator = must copy a's elements

Copy Assignment (Implementation)

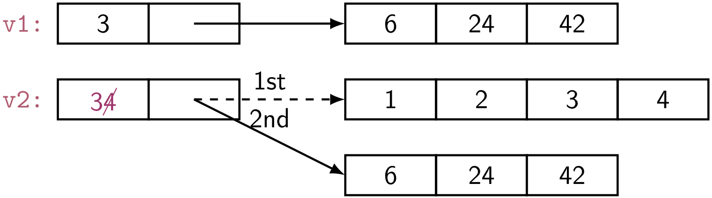
Like copy constructor, but we must deal with old elements.
Make a copy of `a` then replace the current `sz` and `elem` with `a`'s

```
vector& vector::operator=(const vector& a)
{
    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 identifier `this` is a pointer that points to the object for which the member function was called (see par. 17.10).
- It is immutable

Copy with Copy Assignment (Implementation)

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



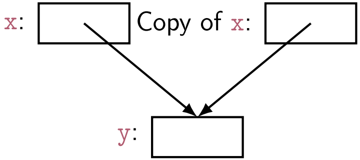
delete[] d by = in previous slide. No memory leak

Operator = must copy a's elements

Copy Terminology

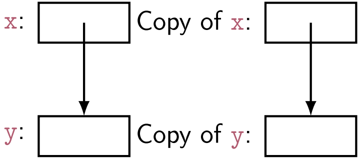
Shallow copy: copy only a pointer so that the two pointers now refer to the same object

- What pointers and references do



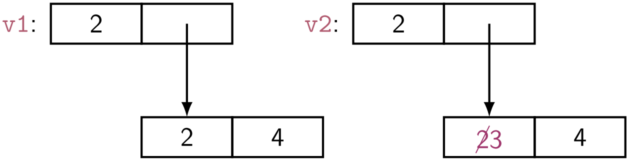
Deep copy: copy what the pointer points to so that the two pointers now each refer to a distinct object

- 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



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
```

Outline

1. Initialization
2. Copy
3. Move
4. Arrays

Move

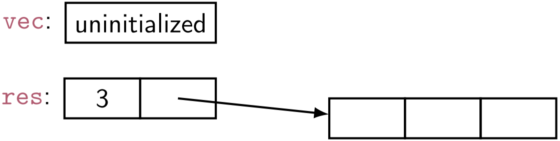
Consider

```
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!
}
```

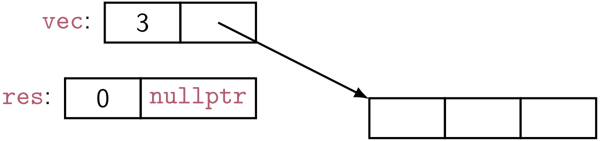
```
void use()
{
    vector vec = fill(cin);
    // ... use vec ...
}
```

Move: What We Want

Before `return res` in `fill()`:



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



Move Constructor and Move 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
    // ...
};
```

&& indicates move

Move Constructor and Assignment (Implementation)

move constructor: “steal” the elements

```
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 assignment: destroy target and “steal” the elements

```
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 (see par. 17.10)
}
```

Essential Operations

- Default constructor
- Constructors from one or more arguments

- 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 these, define them all

Outline

1. Initialization
2. Copy
3. Move
4. Arrays

Arrays

Arrays don't have to be on the free store

```
char ac[7];                    // global array - "lives" forever - in static storage
int max = 100;
int ai[max];

int f(int n)
{
   char lc[20];      // local array - "lives" until the end of scope - on stack
   int li[60];
   double lx[n];    // error: a local array size must be known at compile time
                    // vector<double> lx(n); would work
   // ...
}
```

Address of &

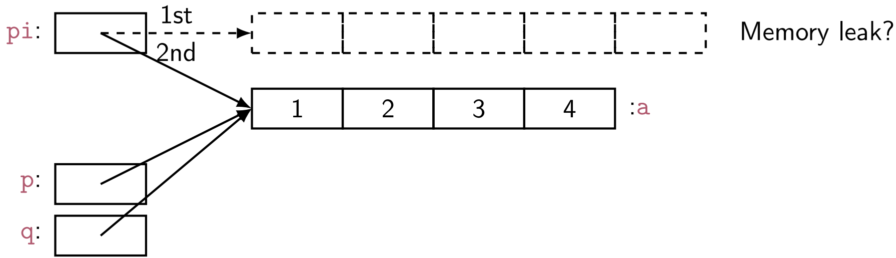
You can get a pointer to any object
not just to objects on the free store

```
int a;
char ac[20];

void f(int n)
{
    int b;
    int* p = &b; // pointer to individual variable
    p = &a;          // now point to a different variable
    char* pc = ac;  // the name of an array names a pointer to its first element
    pc = &ac[0]; // equivalent to pc = ac
    pc = &ac[n]; // pointer to ac's nth element (starting at 0th)
                 // warning: range is not checked
    // ...
}
```

Arrays Convert to Pointers

```
void f(int pi[ ])          // equivalent to void f(int* pi)
{
    int a[ ] = { 1, 2, 3, 4 };
    int b[ ] = a;          // error: copy isn't defined for arrays
    b = pi;   // error: copy isn't defined for arrays. Think of a
              // (non-argument) array name as an immutable pointer
    pi = a;       // ok: but it doesn't copy: pi now points to a's first element
              // Is this a memory leak? (maybe)
    int* p = a;   // p points to the first element of a
    int* q = pi; // q points to the first element of a
}
```



Arrays don't Know Their Size

Warning: very dangerous code, for illustration only: never "hope" that sizes will always be correct

```
void f(char pc[ ], int n) // equivalent to void f(char* pc, int n)
{
    char buf1[200];        // you can't say 'char buf1[n];' n is a variable
    strcpy(buf1,pc);       // copy characters from pc into buf1
                           // strcpy terminates when a '\0' character is found
                           // hope that pc holds less than 200 characters
    // alternative that hedges against pc holding > 200 chars
    strncpy(buf1,pc,200);    // copy 200 characters from pc to buf1
                             // padded if necessary, but final '\0' not guaranteed
}
```

Similarly:

```
void f(int pi[ ], int n) // equivalent to void f(int* pi, int n)
{
    int buf2[300];         // you can't say 'int buf2[n];' n is a variable
    if (300 < n) error("not enough space");
    for (int i=0; i<n; ++i) buf2[i] = pi[i]; // hope that pi really has space for
                                             // n ints; it might have less
}
```

Be Careful with Arrays and Pointers

Watch out on dangling pointers (pointers to deleted memory)

```
char* f()
{
    char ch[20];
    char* p = &ch[90];
    // ...
    *p = 'a';               // we don't know what this will overwrite
    char* q;                // forgot to initialize
    *q = 'b';               // we don't know what this will overwrite
    return &ch[10];         // oops: ch disappears upon return from f()
                            // (an infamous dangling pointer)
}
```

```
void g()
{
    char* pp = f();
    // ...
    *pp = 'c';     // we don't know what this will overwrite
    // (f's ch is gone for good after the return from f)
}
```

Why Bother with Arrays?

- It's all that C has
  - In particular, C does not have `vector`
  - There is a lot of C code "out there"
  - There is a lot of C++ code in C style "out there"
  - You'll eventually encounter code full of arrays and pointers

- They represent primitive memory in C++ programs
  We need them (mostly on free store allocated by `new`) to implement better container types

- Avoid arrays whenever you can
  - They are the largest single source of bugs in C and (unnecessarily) in C++ programs
  - They are among the largest sources of security violations, usually (avoidable) buffer overflows

Recap: Types of Memory

```
vector glob(10);                          // global vector - ‘‘lives’’ forever

vector* some_fct(int n)
{
    vector v(n);                  // local vector - ‘‘lives’’ until the end of scope
    vector* p = new vector(n);    // free-store vector - ‘‘lives’’ until we delete it
    // ...
    return p;
}

void f()
{
    vector* pp = some_fct(17);
    // ...
    delete pp;   // deallocate the free-store vector allocated in some_fct()
}
```

it’s easy to forget to delete free-store allocated objects
so avoid `new/delete` when you can (and that’s most of the time)

Vector: Primitive Access

A very simplified vector of doubles:

```
vector v(10);
```

Pretty ugly access:

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

We're used to this way of accessing:

```
for (int i=0; i<v.size(); ++i) {
    v[i]=i;
    cout << v[i];
}
```

Vector: Pointers for Access

A very simplified vector of doubles:

```
class vector {
    int sz;                    // the size
    double* elem;              // pointer to elements
public:
    explicit vector(int s) :sz{s}, elem{new double[s]} { }        // constructor
    // ...
    double* operator[ ](int n) { return &elem[n]; }  // access: return pointer
};

vector v(10);
```

Access via pointers:

```
for (int i=0; i<v.size(); ++i) {
    *v[i] = i;                 // means *(v[i]), that is, return a pointer to
                               // the ith element, and dereference it
    cout << *v[i];
}
```

It works, but still too ugly.

Vector: References for Access

A very simplified vector of doubles:

```
class vector {
    int sz;                  // the size
    tdouble* elem;           // pointer to elements
public:
    explicit vector(int s) :sz{s}, elem{new double[s]} { }          // constructor
    // ...
    double& operator[ ](int n) { return elem[n]; }  // access: return reference
};

vector v(10);
```

Access via references:

```
for (int i=0; i<v.size(); ++i) {
   v[i] = i;                            // v[i] returns a reference to the ith element
   cout << v[i];
}
```

It works and it looks right!!

Pointer and Reference

You can think of a **reference** as an automatically dereferenced immutable pointer, or as an alternative name (alias) for an object

- Assignment to a pointer changes the pointer's value
- Assignment to a reference changes the object referred to
- You cannot make a reference refer to a different object

```
int a = 10;
int* p = &a;  // you need & to get a pointer
*p = 7; // assign to a through p
        // you need '*' (or '[ ]') to get to what a pointer points to
int x1 = *p;  // read 'a' through 'p'

int& r = a;   // 'r' is an alias for 'a'
r = 9;        // assign to 'a' through 'r'
int x2 = r;   // read 'a' through 'r'

p = &x1;     // you can make a pointer point to a different object
r = &x1;     // error: you can't change the value of a 'r'
```

Summary

1. Initialization
2. Copy
3. Move
4. Arrays