DM560

Introduction to Programming in C++

Vector and Free Store
(Pointers and Memory Allocation)

Marco Chiarandini

Department of Mathematics & Computer Science
University of Southern Denmark

[*Based on slides by Bjarne Stroustrup*]

Outline

1. Pointers
2. Memory Allocation
3. Access
4. Memory Leaks and Destructors
5. void*

Overview

- Vector revisited: How are they implemented?
- Pointers and free store
    - Allocation (`new`)
    - Access
        - Arrays and subscripting: `[]`
        - Dereferencing: `*`
    - Deallocation (delete)
- Destructors
- Initialization
- Copy and move
- Arrays
- Array and pointer problems
- Changing size
- Templates
- Range checking and exceptions

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

Outline

1. Pointers
2. Memory Allocation
3. Access
4. Memory Leaks and Destructors
5. void*

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

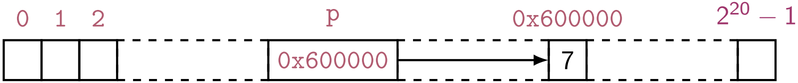
```
// a very simplified vector of doubles (like vector<double>):
class vector {
   int sz;              // the number of elements ('the size')
   double* elem;        // pointer to the first element
public:
   vector(int s);    // constructor: allocate s elements,
                     // let elem point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

Outline

1. Pointers

2. Memory Allocation

3. Access

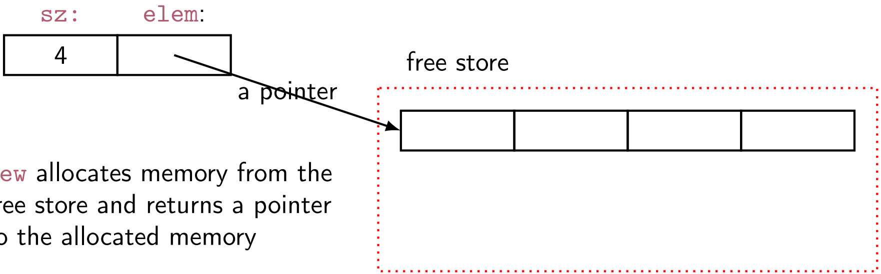
4. Memory Leaks and Destructors

5. void*

Vector: Constructor

An (simplified) implementation of the 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)
```

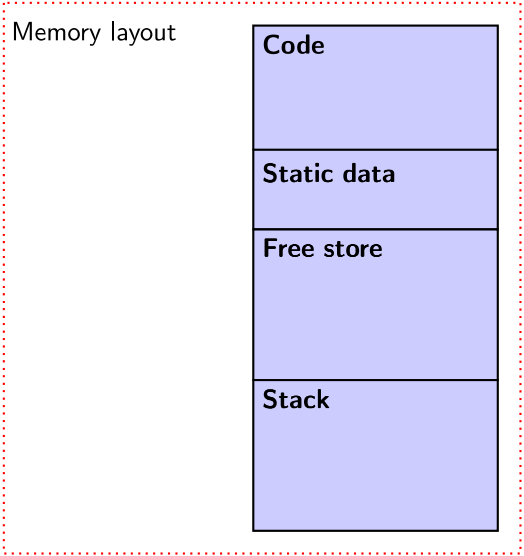


`new` allocates memory from the free store and returns a pointer to the allocated memory

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**



Memory layout

Code

Static data

Free store

Stack

The Free Store (aka 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 ints
                      // "an array of 7 ints"
double* pd = new double[n];     // allocate n uninitialized doubles
```

- A pointer points to an object of its specified type
- A pointer does not know how many elements it points to

Outline

1. Pointers
2. Memory Allocation
3. Access
4. Memory Leaks and Destructors
5. void*

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
```

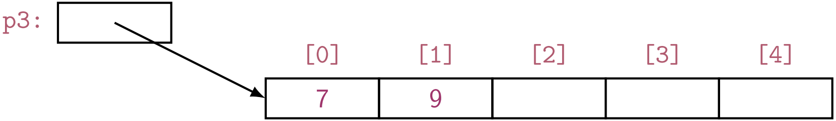
```
int x = *p2;            // get/read the value pointed to by p2
                        // (or "get the contents of what p2 points to")
                        // in this case, the integer 5
```

```
int y = *p1;      // undefined: y gets an undefined value; don't do that
```

Access

Arrays are sequences of elements numbered `[0], [1], [2], ...`:

```
int* p3 = new int[5];     // get (allocate) 5 ints:
```



- set (write to) the 1st element of `p3`

```
p3[0] = 7;
p3[1] = 9;
```

- get the value of the 2nd element of `p3`

```
int x2 = p3[1];
```

- the **dereference operator** `*` for an array: `*p3` means `p3[0]` (and vice versa)

```
int x3 = *p3;
```

Why Use Free Store?

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

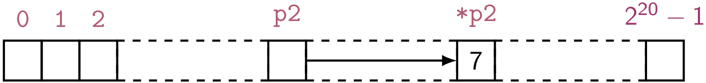
```
double* make(int n)     // allocate n ints
{
    return new double[n];
}
```

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";
cout << "p2==" << p2 << " *p2==" << *p2 << "\n";
```

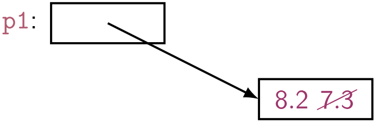
Output:

```
p1==0x7fbba54028b0 *p1==7
p2==0x7fbba54028c0 *p2==7
```

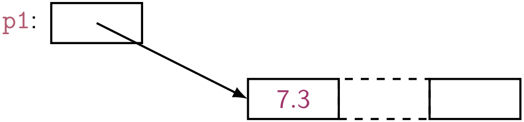
Access

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

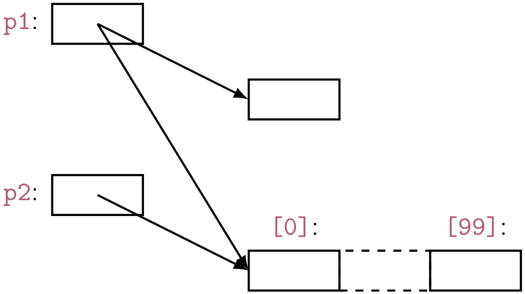
A pointer does not know the number of elements that it's pointing to

```
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
```



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
```

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) << ' '; }
```

Outline

1. Pointers
2. Memory Allocation
3. Access
4. Memory Leaks and Destructors
5. void*

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
    return result;
}

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

- 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

Memory Leaks

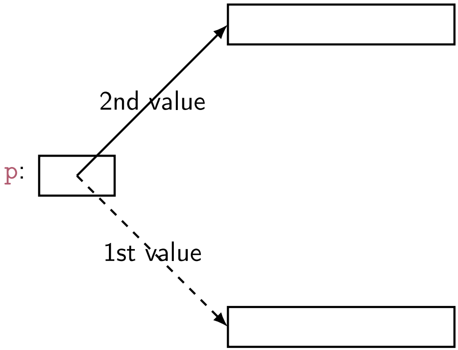
- A program that needs to run “forever” can’t afford any memory leaks
  An operating system is an example of a program that runs “forever”
- If a function leaks 8 bytes every time it is called, how many megabytes it has leaked/lost if it is called 130,000 times?
- All memory is returned to the system 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;
}
```

The 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`
- 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)

Vector: 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

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 deletes 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 = nullptr;   /// old C++ char* p=0;
delete p;       // harmless
```

Outline

1. Pointers
2. Memory Allocation
3. Access
4. Memory Leaks and Destructors
5. void*

`void*`

- `void*` means "pointer to some memory that the compiler doesn't know the type of"

- We use `void*` when we want to transmit an address between pieces of code that really don't know each other's types – so the programmer has to know
  Example: the arguments of a callback function

- There are no objects of type void

```
void v;     // error
void f();   // f() returns nothing
            // f() does not return an object of type void
```

- Any pointer to object can be assigned to a `void*`

```
int* pi = new int;
double* pd = new double[10];
void* pv1 = pi;
void* pv2 = pd;
```

`void*`

To use a `void*` we must tell the compiler what it points to

```
void f(void* pv)
{
   void* pv2 = pv;  // copying is ok (copying is what void*s are for)

   double* pd = pv; // error: can't implicitly convert void* to double*
   *pv = 7;         // error: you can't dereference a void*
                    // this's good: the int 7 is not represented like the double 7.0
   pv[2] = 9;       // error: you can't subscript a void*
   pv++;            // error: you can't increment a void*
   int* pi = static_cast<int*>(pv);      // ok: explicit conversion
   // ...
}
```

- A `static_cast` can be used to explicitly convert to a 'pointer to object type'
- `static_cast` is a deliberately ugly name for an ugly (and dangerous) operation – use it only when absolutely necessary

`void*`

- `void*` is the closest C++ has to a plain machine address
- Some system facilities require a `void*`
- For example, in the callback of the FLTK FUI, `Address` is a `void*`:

```
typedef void* Address;
void Lines_window::cb_next(Address,Address)
```

Pointers and References

Think of a **reference** as:

- an automatically dereferenced **pointer**
- or as "an alternative name for an object" (alias)

Differences:

- a reference must be initialized
- the value of a reference cannot be changed after initialization

```
int x = 7;
int y = 8;
int* p = &x;     *p = 9;
p = &y; // ok
int& r = x;      x = 10;
r = &y; // error (and so is all other attempts to change what r refers to)
```

Summary

1. Pointers
2. Memory Allocation
3. Access
4. Memory Leaks and Destructors
5. void*

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?)

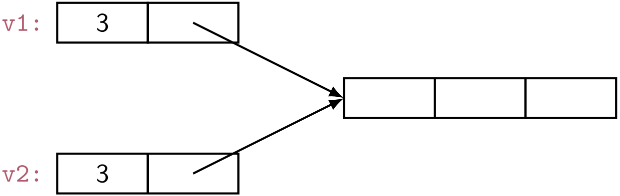
```
void f(int n)
{
    vector v(n);            // define a vector
    vector v2 = v;          // what happens here?
                            // what would we like to happen?
    vector v3;
    v3 = v;                 // what happens here?
                            // what would we like to happen?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

- 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

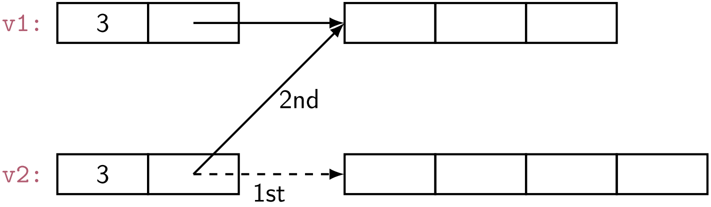
```
void f(int n)
{
    vector v1(n);
    vector v2 = v1;      // initialization:
                          // by default, a copy of a class copies its members
                          // so sz and elem are copied
}
```



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

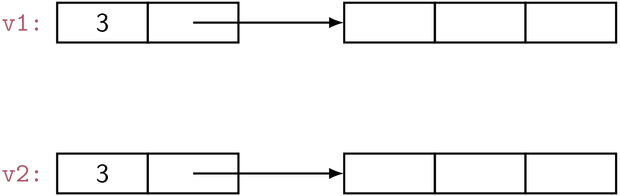
Copy Constructor (Initialization)

```
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
}
```

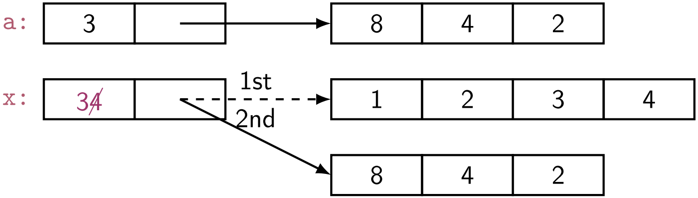


The destructor correctly deletes all elements
(once only for each vector)

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)

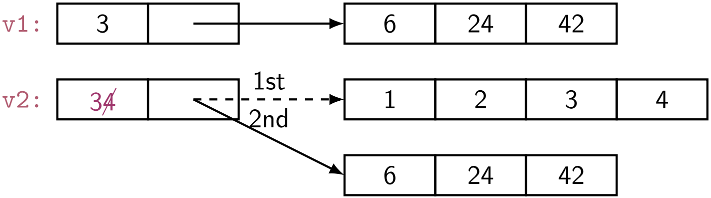
Like copy constructor, but we must deal with old elements.
Make a copy of `a` then replace the current `sz` and `elem` with `a`'s

```
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
}
```



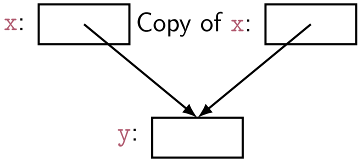
`delete[] d` by `=` in previous slide. No memory leak

Operator `=` must copy `a`'s elements

Copy Terminology

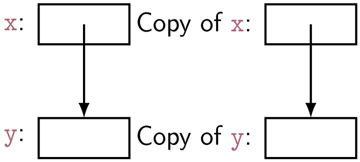
Shallow copy: copy only a pointer so that the two pointers now refer to the same object

- What pointers and references do



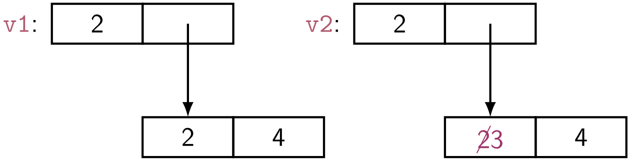
Deep copy: copy what the pointer points to so that the two pointers now each refer to a distinct object

- 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

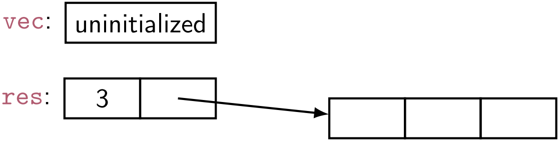
Consider

```
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!
}
```

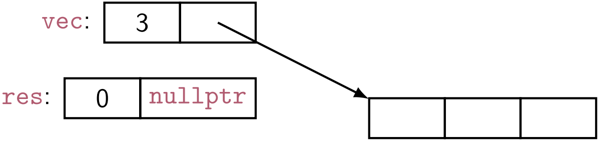
```
void use()
{
    vector vec = fill(cin);
    // ... use vec ...
}
```

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 &

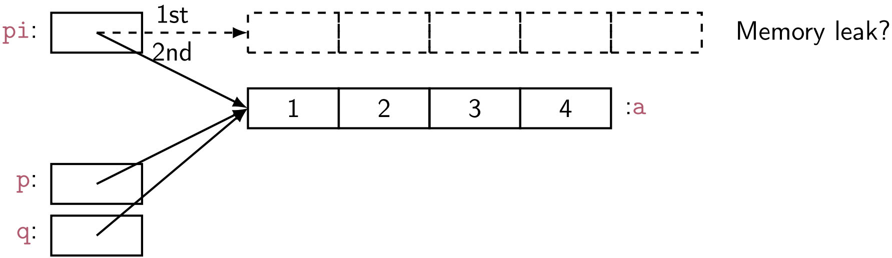
You can get a pointer to any object
not just to objects on the free store

```
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