Software Design (C++)

2. User-defined types in C++ - ADT Programming

Juha Vihavainen
University of Helsinki

Preview

- classes and abstract data types
- class invariants
- orthodox class model: creating and copying values: four essential operations (at least)
- case: implementing a STL-style *custom* **Vector** class
- managing free store (heap), preventing leaks
- lastly, some optimizations

Classes: the idea

- A class directly represents a concept in a program
  - if you can think of "it" as a separate entity, it is plausible that it could be a class or an object of a class
  - examples: vector, iterator, matrix, input stream, string, FFT, valve controller, robot arm, device driver, picture on screen, dialog box, graph, window, temperature reading, clock
- A class is a *user-defined type* that specifies how objects of the type can be created and used (and finally destroyed)
- In C++ (as in most modern languages), a class is the key building block for large programs
  - and useful for small ones also
- The concept was originally introduced in *Simula 67* (Norway)

Classes

A class is a user-defined type: numeric data, container, etc.

```
class X {          // this class' name is X
public:            // public members -- the interface to users
                   // accessible by all
    // functions
    // types
    // data (often best kept private)
private:           // private members -- "implementation details"
                   // accessible by members or friends, only
    // functions
    // types
    // data            . . . perhaps more parts
};
```

struct and *class*

- A **struct** is simply a **class** where members are **public** by default:

```
struct X {
    int m;   . . .
};
```

  means exactly the same as:

```
class X {
public:
    int m;   . . .
};
```

- **struct**s are often used for low-level "technical" data storage
  - "plain data" (some data items just bundled together)
  - the data members can take "any value" (no *invariant*s)

Class *invariants*

- The notion of a “valid instance” is an important special case of the idea of a valid value
- We try to design our types so that values are guaranteed to be valid
  - the state is OK before operations, and left OK after operations
  - checking for validity is an important way to debug code
- A rule for what constitutes a valid value is called **an** *invariant*
  - the invariant for **Date** (“Date must represent a date in the past, present, or future”) is actually a bit complicated
  - remember leap years and Feb 29
    (if divisible by four, except for .. except for .. etc.)
- Try hard to think of good invariants for your classes
  - clarifies design, and may reveal bugs during testing
- If we can’t think of good invariant, we probably have "plain data"
  - if so, can use a **struct** (the standard uses e.g., **pair <*const* Key, T>**)

Classes

Date:

myBirthday:	y	1950
(Bjarne)	m	12
	d	30

```
// simple date
class Date {
public:
    Date (int y, int m, int d);       // constructor checks and initializes
    void addDay (int n = 1);          // increase the date by n days
    int month const ();               // these all are declarations, only
    // . . .
private:                              // we prefer implementation details last
    int y, m, d;                      // year, month, day
};
```

discussed later

.h file

```
Date::Date (int yy, int mm, int dd)        // definition; note :: "member of"
  : y (yy), m (mm), d (dd) { /* . . . */ } // special member initializers

void Date::addDay (int n) { /* . . . */ }  // also a separate definition
```

.cpp file

Classes: checking invariant

What can we do in case of an invalid date?

```
class Date {
public:
    Date (int y, int m, int d);                  // check date and initialize
    // . . .
    static bool check (int y, int m, int d);     // is (y,m,d) a valid date?
private:
    int y, m, d;                                 // year, month, day
};

Date:: Date (int yy, int mm, int dd)
    : y (yy), m (mm), d (dd) {                   // initialize data members
    if (! check (y, m, d))                       // check for validity
        throw std::invalid_argument ("Invalid date: " . . . );
}
```

an initialization list

Classes: public/private distinction

- To provide a clean interface
  - data and messy functions can be made **private**
- To maintain a class invariant (defines valid states for instances)
  - control what can be called outside via **public** functions
  - only the fixed set of functions can access the data
    - member functions and perhaps some **friend**s
- To ease debugging since
  - only the fixed set of functions can directly access private data
    - known as the "*round up the usual suspects*" technique
- To achieve so-called "*representation independence*"
  - allows a change of representation without affecting the clients
  - you need only to modify a fixed set of functions, and these changes to data structures/code don't propagate elsewhere

More on classes

Date:

myBirthday:		
	y	1950
	m	12
	d	30

```
// simple Date using Month type
enum class Month {    // add static type info
    Jan = 1, Feb, Mar, . .  Nov, Dec };
class Date {
public:
    Date (int y, Month m, int d); . . .        // checks for valid date
private:
    int y;                                     // year
    Month m;                                   // month
    int d;                                     // day
};
Date myBirthday (1950, 30, Month::Dec);        // error: 30 is not Month
Date myBirthday (1950, Month::Dec, 30);        // ok
```

Const member functions

- Distinguish between functions that can modify (mutate) objects and those that cannot

```
class Date {
public:   // …
   int day () const { return d; }       // const member: can't modify
   void addDay (int n = 1);             // non-const member: can modify
   // …
};
Date d (2000, Month::Jan, 20);       // can change its state/value
const Date cd (2001, Month::Feb, 21);   // only const functions
std::cout << d.day () << " – " << cd.day () << std::endl;   // ok
d.addDay (1);                                               // ok
cd.addDay (1);                       // error: const violation
```


Summary: a good class interface

- *Minimal*
  - as small as possible - to minimize complexity and testing
- *Complete*
  - but no smaller: provide all "essential" operations (or ways to define them via the given operations)
- *Type safe*
  - let the compiler help and check statically
  - e.g., beware of confusing argument orders
- ***const**-correct*
  - to support **const-**qualified data, objects, or references

Classes: four essential operations

- *<u>Default constructor</u>*; better called: "*zero-argument constructor*"
  - can define yourself: **Date () : y (1), m (Month::Jan), d (1) { . . }**
  - the compiler-generated default implementation calls default member initializations, but only for *class-type* data members
  - <u>no</u> zero-arg. constructor is generated <u>if any other</u> ctors are declared
- *<u>Copy constructor</u>* (defaults to: copy all the data members)
- *<u>Copy assignment</u>* (defaults to: copy all the data members)
- *<u>Destructor</u>*
  - can define yourself: **~Date () { /*** *release resources - if any* ***/ }**
  - the default one calls destructors for *class-type* data members, only

For **Date**, the default implementations (happen to) work OK

```
Date d;          // ok: default constructor . . (we assume)
Date d2 = d;     // ok: copy initialized (copies the members)
d = d2;          // ok: copy assignment (copies the members)
```


C++ example: *IntStack*

```
class IntStack {                              // in a header file (.h)
public:
    explicit IntStack (size_t sz = 90);       // not a size_t conversion
    void push (int);
    int pop (); . . .
    IntStack (IntStack const&);               // copy constructor
    IntStack& operator = (IntStack const&);   // assignment (value copy)
    ~IntStack ();                             // destructor
private:          note the overload of the operator =
    size_t size_;                             // maximum capacity
    size_t top_;                              // current number of items
    int * array_;                             // int array for items
};
```

Note. ***size_t*** is (ANSI C) unsigned type (result of **sizeof** operator).

IntStack (continued)

```
IntStack::IntStack (size_t sz)                  // placed in a .cpp file
    : size_(sz), top_(0), array_(new int [sz]) {   // no explicit needed here
}                       // always initialized in the order of the declarations
void IntStack::push (int i) {
    if (top_== size_)  throw std::logic_error ("stack overflow");
    array_[top_++] = i;
}
IntStack::IntStack (IntStack const& stack)
    : size_(stack.size_), top_(stack.top_), array_(new int [size_]) {
    for (size_t i = 0; i < top_; ++i) array_[i] = stack. array_[i];
}
IntStack::~IntStack  () {
    delete [] array_;                           // note the brackets [] !
}
```


Interfaces and "helper" functions

Keep a class interface (the set of public functions) minimal to

- simplify understanding
- simplify debugging
- simplify maintenance

Keeping the class interface minimal, may require extra "helper" functions outside the class (i.e., non-member functions)

- e.g., overloaded == (equality) , **!=** (inequality)
- **nextWeekday (), nextSunday ()** (see next slide)

Note. No comparison operators are defined by default.

Sample "helper" functions

```
Date nextSunday (Date const& d) {
    // access d using d.day (), d.month (), and d.year ()
    // construct a Date to return
}

Date nextWeekday (Date const& d) { /* . . . */ }

bool operator == (Date const& a, Date const& b) {
    return a.year () == b.year () &&
           a.month () == b.month () &&
           a.day () == b.day ();
}
// we must also define (since not generated by default):

bool operator != (Date const& a, Date const& b) { return !(a==b); }
```


Remember to support access to *const* objects

```
class IntStack {
    // . . .
    int& operator [] (std::size_t n);        // access n'th item (from top)
    int operator [] (std::size_t n) const;   // don't allow updates
    // . . .
};
IntStack a;
IntStack b;
// push ints on a and b….
f(a,b); // call f, a and b will not be copied, but accessed via reference
void f (IntStack const& cstack, IntStack& stack) {
    // . . .
    int i1 = cstack [7];           // call sthe const version of []
    int i2 = stack [7];            // calls the non-const version of []
    cstack [7] = 9;                // error: calling the const version of []
    stack [7] = 9;       . . .     // ok : non-const version of []
}
```


Note. If **const**-methods are missing, cannot manipulate **const** data.

Container services

Switching contents of two objects safely:

s1.swap (s2); // *swap contents safely and efficiently* (*O* (1))

- often utilised for safe updates: build a new version, then **swap**

Access to elements via *iterators* (very efficient but <u>*sometimes*</u> unsafe)

```
class Vector { . . .                       // hypothetical Vector class
public:
   typedef double * iterator;              // often implemented as pointers
   iterator begin () { return elem; }      // the first element (if any!)
   iterator end () { return elem + size (); }   // beyond the last item
};
```

for an empty container: **begin () == end ()**

C++ standard does not (directly) support IO for containers (vs. Java/C#)

- e.g., to print out a container as a whole value
- but you can provide such operations yourself: discussed later on

Summary: to Construct() - or not?

- Does my class need a (default) zero-argument constructor?
  - Yes, if you need to be able create instances of the class without any initializers: vector<My_class> vec(10);
  - Requires that you can establish the *invariant* for the class with a meaningful and obvious *default value*
- Does my class need a destructor?
  - Yes, if it has *acquired pointers or references* to dynamically allocated objects or other resources (e.g. a database session) that need to be properly disposed of to avoid wasting them
- If your class needs a destructor, it most likely also needs:
  - Copy constructor, (copy) assignment, move constructor (C++11), and move assignment (C++11)

Vector revisited

- Using pointers and free store (overview/refreshment)
  - allocation with operators: **new** and **new []**
  - deallocation with operators **delete** and **delete []**
  - access: arrays, subscripting **[]**, and dereferencing: *****
- Destructors (more implementation details)
- Definition and use of copy constructor and copy assignment
- Move constructor and move assignment (new in C++11)
- C-style arrays and potential problems with pointers
- Making the size of a container flexible (**resize**)

Why look at a *Vector* implementation?

- To see how the standard containers "really" work (almost)
- To learn the common properties of standard containers
- To introduce basic concepts and language features
  - "free store" (heap), copying, and dynamically growing data structures
- To see how to directly deal with raw memory
  - and how to (mostly) hide that from clients
- To see techniques and concepts you still need from C
  - including the dangerous ones
- To demonstrate basic class design techniques
- To see some essential techniques and good design

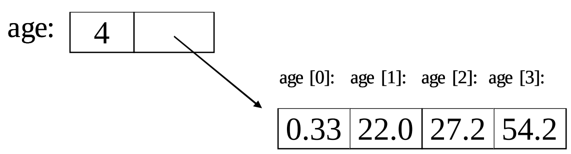
Building from the ground up

- The hardware provides memory and addresses
  - low level, untyped, fixed-sized, no checking of access
  - as fast as hardware architectures can make it
  - pointers and address arithmetics (directly from C)
- *Java* and other object-oriented languages are build on the top of *VMs*
- The application programmer needs something like a **Vector**
  - statically type checked (well mostly)
  - size is flexible (grows dynamically as we get more data)
  - run-time checking (often optionally, or for "debug" versions)
  - very close to optimally fast
- The techniques for building **Vector** are the ones underlying all designs of similar C++ data structures

A custom ***Vector*** class

- Can hold an arbitrary number of elements
  - up to whatever physical memory and the operating system can handle
- Number of elements can vary over time (at least, for the final version)
  - e.g. by using **push_back()**
- For example

Vector age (4); // *4 items* (*maybe initialized to zero*)
Age [0] = .33; age [1] = 22.0; age [2] = 27.2; age [3] = 54.2;



First, a very *simplified Vector*

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

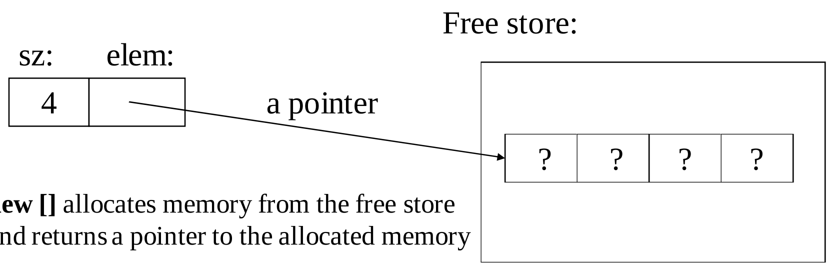
Stroustrup systematically uses **int** for size type (another alternative would be **std::size_t**)

Perhaps for invariant checks.. (sz >= 0)

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
}                               // and store a pointer in elem
```

Note 1: **new[]** *does not initialize primitive elements* (*but* **std::vector** *does*)
Note 2: **new[]** *initializes elements of class-type* (*using 0-argument ctor*)



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

The computer's memory (conceptual)

- The executable code are in “the code section”
- Global variables are “static data” (constructed before **main()**)
- Local variables “live on the stack” (function call stack)

(hypothetical)

- Static objects in different translation units are initialized in undefined order.

memory layout:

- The static area is initialized to zero.
- Heap objects are created by **new** and destructed by **delete**.
- Temporaries are managed by the compiler/system

- Local objects (variables) are automatically destructed by the run-time system: implicit destructor calls at the end of the block.

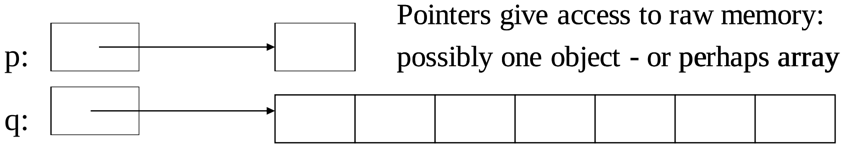
The free store ("the heap")

- we 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

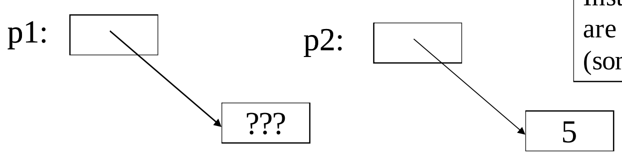
int * p = new int;	// *allocate one uninitialized* **int**
int * q = new int [7];	// *allocate seven uninitialized* **int**s
	// *“an array of* 7 **int**s*”*
double * pd = new double [n];	// *allocate* **n** *uninitialized* **double**s

- a pointer points to an object of its specified type, but
- a pointer *never* knows how many elements it points to => *errors*!

Pointers give access to raw memory: possibly one object - or perhaps array



Access to dynamic objects



Instances of classes are always initilized (somehow..)

Individual elements

```
int * p1 = new int;          // get (allocate) a new uninitialized int
int * p2 = new int (5);      // get a new int initialized to 5
                             //     note the constructor-style syntax

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

int y = *p1;                 // y gets an undefined value
```


A problem: *memory leaks* (with C-style code)

```
double * calc (int result_size, int max) {  // illustrative, only
   int * p = new int [max];        // max ints from the free store
   double * result = new double [result_size];
   // . . . use p to calculate results to be put in result . . .
   delete [] p;                    // free that array (if you remember)
   return result;
} . . .

double * res = calc (200, 100);
// use res . . here
delete [] res;                     // again, easy to forget
```

Question: *What if the above code throws exceptions?*

On memory leaks

- At the end of a program, its memory is returned to the system
- A program that "runs forever" can't afford any memory leaks
  - an operating system is a program that may "run forever"
  - depending on circumstances, need to *recycle* memory space
- A program that runs to completion with predictable memory usage *may* "*leak*" without causing problems
  - i.e., **new** operations without corresponding **delete** operations
  - so, memory leaks aren't "good/bad" but can be a problem in specific circumstances
- By default, better to **delete** to make behavior predictable *and* testable; makes easier to *track memory* during debugging

How to avoid memory leaks

Messing directly with **new** and **delete** is tedious and dangerous

- better to use a well-behaving data type, such as **vector**, **list**, etc.
  - or define such a container yourself . .

Sometimes, we might use a custom *garbage collector*

- reclaiming unused memory without relying on user-supplied **delete** or *free* commands; a permitted but not required technique for C++
- a program the keeps track of allocations and periodically returns unused free-store allocated memory to the free store
  - see e.g. Hans-J. *Boehm's garbage collector* (*Boehm* GC) for C/C++; http://www.hpl.hp.com/personal/Hans_Boehm/gc/
- a garbage collector doesn’t prevent all leaks - may have valid but unused links and data ("garbage"); uses safe "conservative" strategy

Vector *destructor*

// a very simplified **Vector** *of* **double**s*:*

```
class Vector {
public:
    explicit Vector (int s)              //  constructor: acquires memory
      : sz (s), elem (new double [s]) {}
    ~Vector () { delete [] elem; }   . . . //  destructor: releases memory
private:
    int sz;                              //  the size
    double * elem;                       //  a pointer to the elements
};
```

An example of a general and important technique (RAII)

- acquire resources in a constructor (called by application code)
- release them in the destructor (called by the run-time system)
- resources can be: memory, files, locks, threads, socketsd,..
- discussed more later on

Solving a problem: memory leaks

```
void f (int x)  {
    int * p = new int [x];     //  allocate x ints  -  bad style!
    Vector v (x);              //  ok: define a Vector (allocates x doubles)
    // . . .  use p and v  . . .
    delete [] p;               //  deallocate p
    //  the memory for v is implicitly deleted here by vector's destructor
}
```

- The **delete** now looks verbose and ugly
  - how can we avoid forgetting such **delete**s in code?
    - experience shows that deletions are easy to forget
    - if interleaving code throws an exception => *leak*
- So, always prefer **delete**s placed inside destructors

Free store: Summary

- **new** allocates an object on the free store, sometimes initializes it, and returns a pointer to it
  - **int * pi = new int;** // *no initialization* (*for* **int**)
  - **char * pc = new char ('a');** // *explicit initialization*
  - **double * pd = new double [n];** // *allocate uninitialized array*
    - **new** throws a **std::bad_alloc** if it can't allocate (for too large **n**)
- **delete** and **delete []** return the memory of an object allocated by **new** to the free store to be used for new allocations
  - **delete pi;** // *deallocate an individual object*
  - **delete pc;** // *deallocate an individual object*
  - **delete [] pd;** // *deallocate an array!*
- **delete** of a null pointer does nothing
  - **char * p = nullptr;** // **nullptr** *was introduced by* C++11
  - **delete p;** // *no warning* (*and harmless*)

What have we got this far?

// *a very simplified* **Vector** *of* **double**s (*as far as we got before*):

```
class Vector {
public:
    explicit Vector (int s) : sz (s), elem (new double [s]) {}
    ~Vector () { delete [] elem; }
    int size () const { return sz; }
      . . .
private:
    int sz;                    // the size
    double * elem;             // pointer to elements
};
```

The compiler thinks this is OK - it isn't!
Something is still missing.

A problem: how are values copied?

Copy doesn't work yet as we would hope, for example

```
void f (int n) {
    Vector v (n);     . . .  //  define a Vector of size n
    Vector v2 = v;           //  copy ctor: initialize with a value (ok?)
    Vector v3;        . . .
    v3 = v;                  //  assign: make a copy of a value (ok?)
    // . . .
}
```

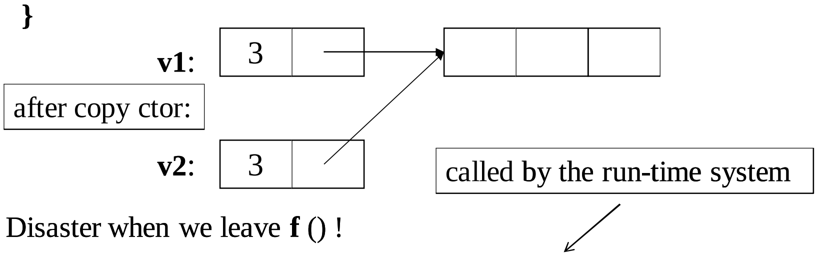
The compiler *generates* missing copy ctor and assignment ("=").

Ideally: **v2** and **v3** get copies of the value of **v** (i.e., "=" copies state), and all memory is returned to the free store upon exit from **f** ()

That's what the standard **std::vector** does but doesnt't happen for our still-too-simple **Vector**.

Naïve copy initialization (the default)

```
void f (int n) {              //  CASE  I
    Vector v1 (n);  . . .     //  these both are initializations
    Vector v2 = v1;           //  by default, a copy just duplicates members
                              //      so sz and elem are copied, only
}
```



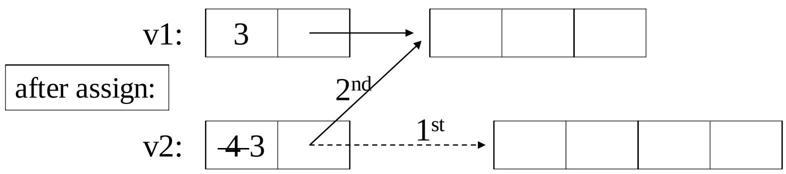
Disaster when we leave **f** () !

v1's elements are *deleted twice* by the destructor
=> undefined behavior - - *possibly* crashes the program

Note that the compiler or the run-time system do not warn about this.

Naïve copy assignment (the default)

```
void f (int n)  {            // CASE II
    Vector v1 (n);  . . .
    Vector v2 (4);  . . .
    v2 = v1;                 // default assignment:  copies members as such
                             //     so, sz and elem are copied
}
```



Again, disaster when leaving **f** ()!

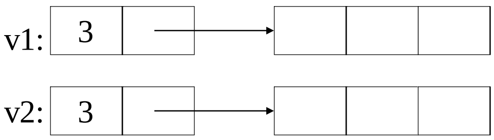
v1's elements are *deleted twice*, again by the destructor;
also memory *leak*: v2's old element array *is not deleted*

Defining a proper copy constructor

```
class Vector {
public:
    Vector (Vector const&);   // defines how to build a new copy
    // . . .
private:
    int sz;
    double * elem;
};
Vector::Vector (Vector const& a)
  : sz (a.sz),  elem (new double [a.sz]) {              //  allocate space
    for (int i = 0; i < sz; ++i) elem [i] = a.elem [i];  //  copy elements
}
```

Using a copy constructor

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



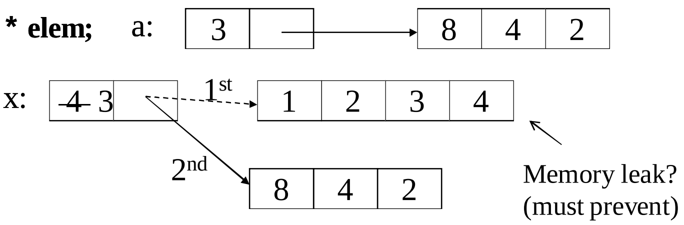
Now OK: the destructor correctly deletes all elements (once only).

- other ctor uses: passing/returning values, temporaries, etc.:

  g (n+2, v1, Vector (v1) , Vector (100)); // *passing value copies*

Next: copy assignment

```
class Vector {
public:
    Vector& operator = (Vector const& a);   //  define how to assign
    // . . .
private:
    int sz;
    double * elem;
};
 . . .
 x = a;
```



operator = needs to copy a's element values (here **double**s)

- this copying is done "recursively" for class-type data members
- e.g., consider: **std::vector <std::vector <std::string>>**

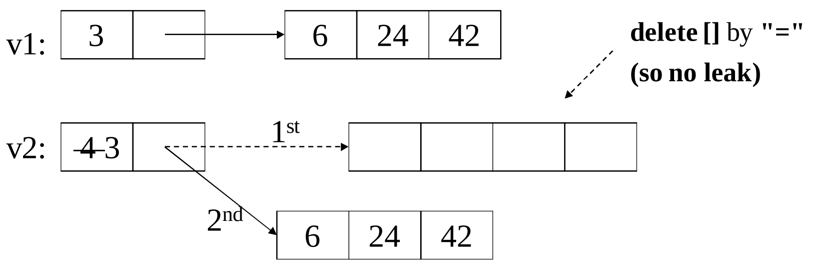
Copy assignment

```
Vector& Vector::operator = (Vector const& a) {
    // like copy constructor, but we must deal with the 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)       // copy elements
        p [i] = a.elem [i];
    delete [] elem;                      // deallocate old space
    elem = p;                            // set new elements
    sz = a.sz;                           // set new size
    return *this;          // by C convention, assign returns a reference
}                          // - could define "void operator= (Vector . . "
```

- The general idea: every data structure manages its elements *and* their memory reservations.

Using copy assignment

```
void f (int n) {
    Vector v1 (n); . . .   // initialize with element values: 6 24 42
    Vector v2 (4); . . .
    v2 = v1;               // assign new contents
     . . .
}                          // v1, v2 are destructed by the system
```



Sometimes we just want to *move* things - not copy them

```
Vector fill (istream is) {
    Vector res;           // define a local Vector to hold input
    for (double x;is >> x;) res.push_back(x);  // read till the end
    return res;   // res is copied using copy constructor to create a
}                 // new object; then res is deleted and cannot be
                  // accessed by anybody anymore

void use()
{
    Vector vec = fill(cin);  // Copy constructor copies the  data
                             // to a new memory buffer in vec
    // …
}
```


Copying can be expensive…

- Can we just somehow ’steal’ the memory reserved by the local variable **res** for holding the data?
  - Suck its ”brains” and leave it to die, since it’s not going to do anything anymore (after the **return** from the function)
- Yes, we can!
  - By defining a *move constructor* for Vector we get just that

Called ”*rvalue reference*”

```
Vector::Vector(Vector&& a)
        :sz{a.sz}, elem{a.elem}   // take a’s elem ands sz
  {
        a.sz = 0;                 // make a the empty vector
        a.elem = nullptr;
  }
```

Moving in action

```
vector fill(istream& is)
{
    vector res;
    for (double x; is>>x; ) res.push_back(x);
    return res;
}
```

- The move constructor is implicitly used to implement the return
  - Compiler knows that the local value returned (**res**) is about to go out of scope
  - Compiler can (generate code to) move from **res**, rather than copy it

Moving works on assignment, too!

```
Vector& Vector::operator=(Vector&& a) // move a to this vector
{
    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
}
```

- The <u>programmer must tell</u> the compiler when move assignment can be used: a_vec = std::move(another_vec);
  - std::move(x) means “give me an rvalue reference to x”

What about C arrays?

- Avoid primitive C arrays whenever you can
  - Stroustrup: the largest single source of bugs in C and (unnecessarily) in C++ programs
  - among the largest sources of security violations -- usually (avoidable) buffer overflows
- It's all that C has; in particular, does not have **vector**s
  - there is a lot of C code "out there"
  - there is a lot of C++ code in C style "out there"
  - may encounter code full of pointers and C arrays
- C arrays should only be used to represent primitive memory
  - mostly allocated on free store by **new**
  - but we still need them to **implement** container types

Accessing *Vector* elements (*Java* style)

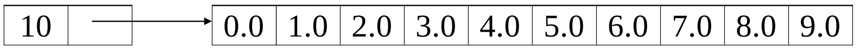
```
// a preliminary simplified Vector of doubles:
class Vector {
public:
   explicit Vector (int s) : sz (s), elem (new double [s]) {}    // ctor
   double get (int n) const { return elem [n]; }            // access: read
   void set (int n, double v) { elem [n] = v; }     . . .     // access: write
private:
   int sz;    // the size
   double * elem;
}; . . .
Vector v (10);
for (int i = 0; i < v.size (); ++i) { v.set (i, i); cout << v.get (i) << ' '; }
```

indexing works for primitive arrays;
note the two differing semantics

10	→	0.0	1.0	2.0	3.0	4.0	5.0	6.0	7.0	8.0	9.0

Vector: access of elements

```
//  a simplified Vector of doubles:
Vector v (10);
for (int i = 0; i < v.size (); ++i) {
    v.set (i, i);
    std::cout << v.get (i) << std::endl;   //  ugly ☹
}
for (int i = 0; i < v.size (); ++i) {
    v [i] = i;                             //  we're used to this
    std::cout << v [i] << std::endl;
}
```



Reminder: pointer vs. reference

- A reference is an automatically dereferenced immutable pointer
- An alternative name for an object (alias)
- A reference must always be initialized -- and is never null..
- We cannot make a reference refer to a different object
- Assignment to a pointer changes the pointer's value
- Assignment via a reference changes the object referred to

```
int a = 10;
int * p = &a;         //  need & to get a pointer
*p = 7;               //  assign to a through p
                      //  need * (or [] ) to get to what a pointer points to
int x1 = *p;          //  read a through p
---------------------------------------------------------------
int& r = a;           //  r is a synonym for a
r = 9;                //  assign to a through r
int x2 = r;           //  read a through r

p = &x1;              //  ok: make a pointer point to a different object
r = &x1;              //  error: you can't change a reference itself
```

Vector: use *references* for access

lvalue in C terms

```
// a simplified Vector of doubles:
class Vector {
public:
    explicit Vector (int s) : sz (s), elem (new double [s]) {}    // . . .
    double& operator [] (int n) { return elem [n]; }    // returns reference
private:
    int sz;                                             // the size
    double * elem;                                      // ptr to elements
};

Vector v (10);
for (int i = 0; i < v.size (); ++i) {          // now works and looks right
    v [i] = i;                                 // v [i] returns a reference to i'th
    std::cout << v [i] << std::endl;           // the same but here gets the value
}
```

10	→	0.0	1.0	2.0	3.0	4.0	5.0	6.0	7.0	8.0	9.0

Continuing on *Vector*

```
// an almost real Vector of doubles:
class Vector {
public:
    // special constructor:
    explicit Vector (int s)         // not a type conversion
        : sz (s), elem (new double [s]), space (s) { . . . }
    // access: returns reference
    double& operator [] (int n) { return elem [n]; }
    int size () const { return sz; }   // current size
    // . . .
    // the four essential operations:
    Vector () : sz (0), elem (nullptr), space (0);
    Vector (Vector const&);
    Vector& operator = (Vector const&);
    ~Vector () { delete [] elem; }
private:
    int sz;           // the size
    double * elem;    // a pointer to the elements
    int space;        // size + free_space (total capacity)
};
```

We want new services

- changing vector size
- representation changed to include *free space*
- adding space management
  - **push_back (double d)**
  - **resize (int n)**
  - **reserve (int n)**
- the **this** pointer
- optimized copy assignment

Changing *Vector* size

- Abstractions that can change size are very convenient
  - e.g., a **Vector** where we can change the number of elements
- How do we create the illusion of change?

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

  we can change its size in three ways (at least)

  - resize it

    v.resize (10); // **v** *now has* **10** *elements* (*somehow*)

  - add an element

    v.push_back (7); // *add* **7** *to the end of* **v**
    // **v.size ()** *increases by* **1**

  - assign to it

    v = v2; // **v** *is now a copy of* **v2**, and
    // **v.size ()** *now equals* **v2.size ()**

- The *standard* **std::vector** provides: **clear()**, **erase()**, **insert()** . .

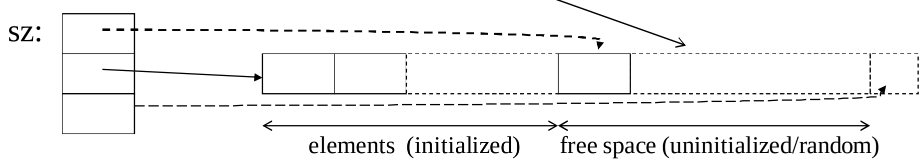
Representing *Vector*

If you **resize()** or **push_back()** once, you'll *probably* do it again

- so let's keep a bit of free space for future expansion

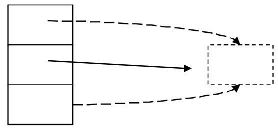
```
class Vector {   //  . . .
private:
    int sz;
    double * elem;
    int space;          // number of elements plus "free space" =
                        //    the number of  value "slots" in the buffer
};
```

can control allocation by: **reserve (minCapacity)**



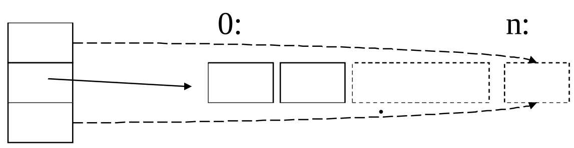
Representing *Vector*

- An empty **Vector ()** (here, no free store use):



So, an empty vector has no dynamic allocation.

- A **Vector (n)** (here, no free space):



Vector::*reserve* ()

First deal with allocation of space; given space all else is easy

- **reserve**() doesn't "mess" with size or element values

```
void Vector::reserve (int newAlloc)  {          //  required min capacity
    //  make the vector have space at least for newAlloc elements
    if (newAlloc <= space)
        return;                                 //  never decrease
    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 (if any!)
    elem = p;  space = newAlloc;                //  new ones into place
                                                //  sz is not changed
}
```

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 any 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 any)
   sz = newsize;               // may be smaller than old size!
}
```

- if the size is decremented, old values are here "left" in their place
- this doesn't matter for values of *primitive types* (**double**, **int**)
- for values of class-types T, we must here actually call the destructor:

  elem [i].T::~T() (can be *potentially dangerous*!)

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 (space == 0)             // no space
      reserve (8);             //   so grab some, say 8
   else if (sz == space)       // space is filled: get more space
      reserve (2 * space);     // double the available space
   elem [sz] = d;              // add d at end
   ++sz;                       // and increase the size
}
```

- doubling the space may avoid some future copying - or we can use **reserve**() to exactly determine the buffer size (and prevent unnecessary reallocations)

Almost real *Vector* of doubles

```
class Vector {                                         // Vector of double
public:
    explicit Vector (int s)                            // constructor
      : sz (s), elem (s?new double [s]:nullptr), space (s) {}
    double& operator [] (int n) { return elem [n]; }   // access item: return reference
    int size () const { return sz; }                   // current size

    void push_back (double d);                         // add new element
    void resize (int newsize);                         // grow (or shrink) size

    void reserve (int minCapacity);                    // get more space (if necessary)
    int capacity () const { return space; }            // current total buffer space
     . . .
    Vector () : sz (0), elem (nullptr), space (0) { }  // zero-arg. (default) constructor
    Vector (Vector const&);                            // copy constructor
    Vector& operator = (Vector const&);                // copy assignment
    ~Vector () { delete [] elem; }                     // destructor
public:
    int sz;                                            // the size (or use size_t)
    double * elem;                                     // a pointer to the elements
    int space;                                         // size + free space
};
```

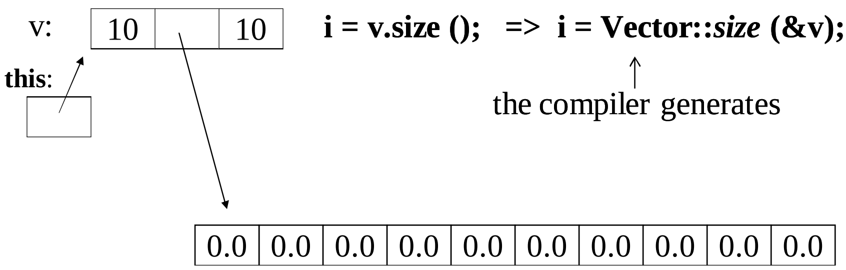

The **this** pointer (~Java, C#)

A *Vector* is an *object* (doesn't matter how it is allocated)

- **Vector v (10);** // *declared object* (*static or local*)
- **Vector * p = &v;** // *we can point to a* **Vector** *object*

p:

- **Vector**'s member functions need also to refer to that object
  - is passed as an *implicit hidden* parameter **this** ("pointer to self") to each (non-**static**) function member



i = v.size (); => i = Vector::*size* (&v);

↑ the compiler generates

The **this** pointer and self reference

By convention, an assignment returns a reference to its object:

```
Vector& Vector::operator = (Vector const& a) {
    // like copy constructor, but deal with old elements . .
    return *this;
}

void f (Vector& v1, Vector& v2, Vector const& v3) {
    // . . .
    v1 = v2 = v3;   // made possible by operator=() returning *this
    // . . .           note that "=" associates to the right
}
```

- **this** pointer has, of course, many more other relevant uses, e.g., when passing objects as operands (or arguments) inside member functions

Reminder: assignment

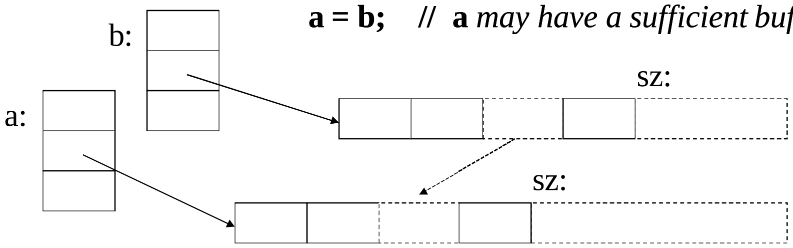
- *copy-and-swap* is a powerful general idea
- e.g., assignment can be implemented as follows

```
Vector& Vector::operator = (Vector const& a) {
    // like copy constructor but we must also 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];     // first, allocate new space
    for (int i = 0; i < a.sz; ++i)      // then, copy elements
        p [i] = a.elem [i];
    delete [] elem;                     // deallocate old space
    elem = p;                           // set new elements
    space = sz = a.sz;                  // set new size & capacity
    return *this;                       // return a self-reference
}
```

To optimize assignment

- Such “copy and swap” is the most general way
  - but not always the most efficient
  - what if there already is sufficient space in the target vector?
    - then leave the buffer alone and just copy element values
    - for example, consider an assigment:

a = b; // **a** *may have a sufficient buffer*



A (more) optimized *Vector* assignment

```
Vector& Vector::operator = (Vector const& 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;                    // change size but don't change capacity
        return *this;
    }
    // otherwise: "make copy and swap"
    double * p = new double [a.sz];       // make new version, may throw
    for (int i = 0; i < a.sz; ++i) p [i] = a.elem [i];
    delete [] elem;                // after successful copy, do safe replacement
    elem = p;  space = sz = a.sz;  return *this;
}
```

Question: *What happens if no check for self-assignment?*
Whether *self-assignment check* is needed, depends on the circumstances.

Summary: Defining user types

- Class invariants define valid states for instances
- C++ doesn't necessarily (by default) initialize data members or elements => it is the programmer's responsibility
- C++ doesn't necessarily (by default) release resources => it is the programmer's responsibility
- Classes with dynamic resources need to define the correct semantics for copying values and to release such resources
- Libraries provide ready-made abstractions with guaranteed initialization and resource management (discussed later..)
- For library classes, may need to optimize away unnecessary overheads (or the programmer may want to write her own)