

Exercise 12 –Classes, Dynamically Allocated Memory

Informatik I für Mathematiker und Physiker (HS 2015)

Yeara Kozlov

Slides courtesy of Kaan Yücer & Endri Dibra

Agenda

- ◆ Homework
- ◆ Classes
- ◆ Dynamically Allocated Memory
- ◆ Dynamic Data Structures
- ◆ Dynamic Storage

Agenda

- ◆ Homework
- ◆ **Classes**
- ◆ Dynamically Allocated Memory
- ◆ Dynamic Data Structures
- ◆ Dynamic Storage

Classes - general remarks

- Classes exist only in C++, not in C
- Classes (**class**) consists of:
 - ◆ Set of *member* variables (like **struct**)
 - ◆ Set of *member* functions (so called *methods*)
 - ◆ Defines visibility of *members* (**public** / **private**)
- Classes vs. structs:
 - ◆ In **classes**, member access is **private**, if not otherwise specified
 - ◆ In **structs**, member access is **public**, if not otherwise specified

Classes - data members

- ◆ Task: define a class for complex numbers:
- ◆ What data members should it have?

Complex Numbers (Recap)

Complex Numbers – Basics

- Imaginary unit i :

$$i \cdot i = -1 \quad \Rightarrow \quad i = \sqrt{-1}$$

Complex Numbers – Basics

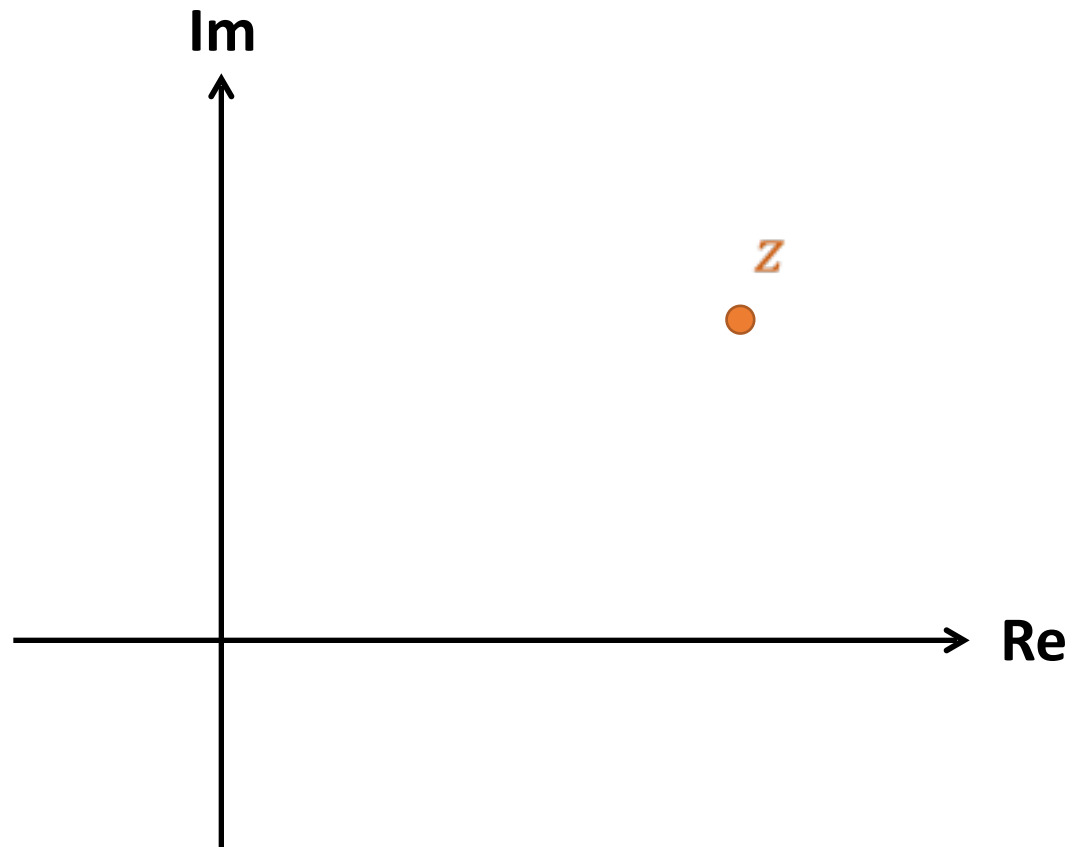
- Imaginary unit i :

$$i \cdot i = -1 \implies i = \sqrt{-1}$$

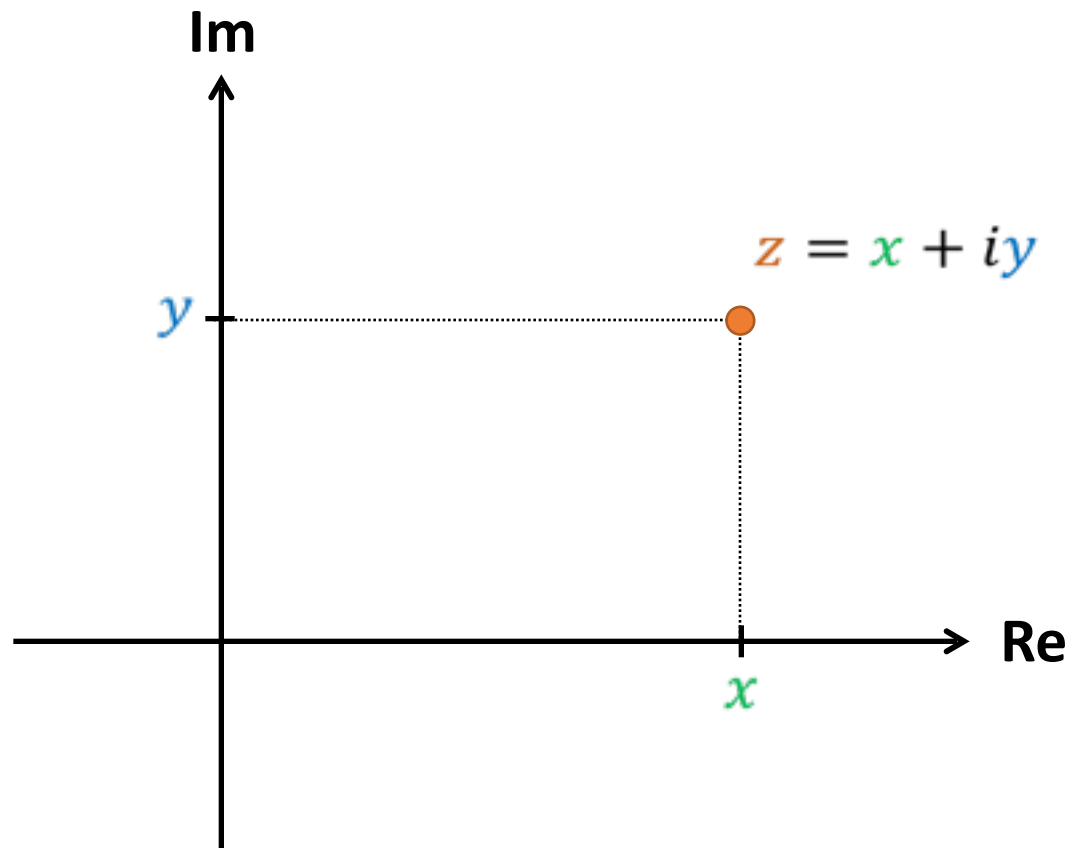
- $z \in \mathbb{C}$ can be represented as

$$z = x + iy \text{ for } x, y \in \mathbb{R}$$

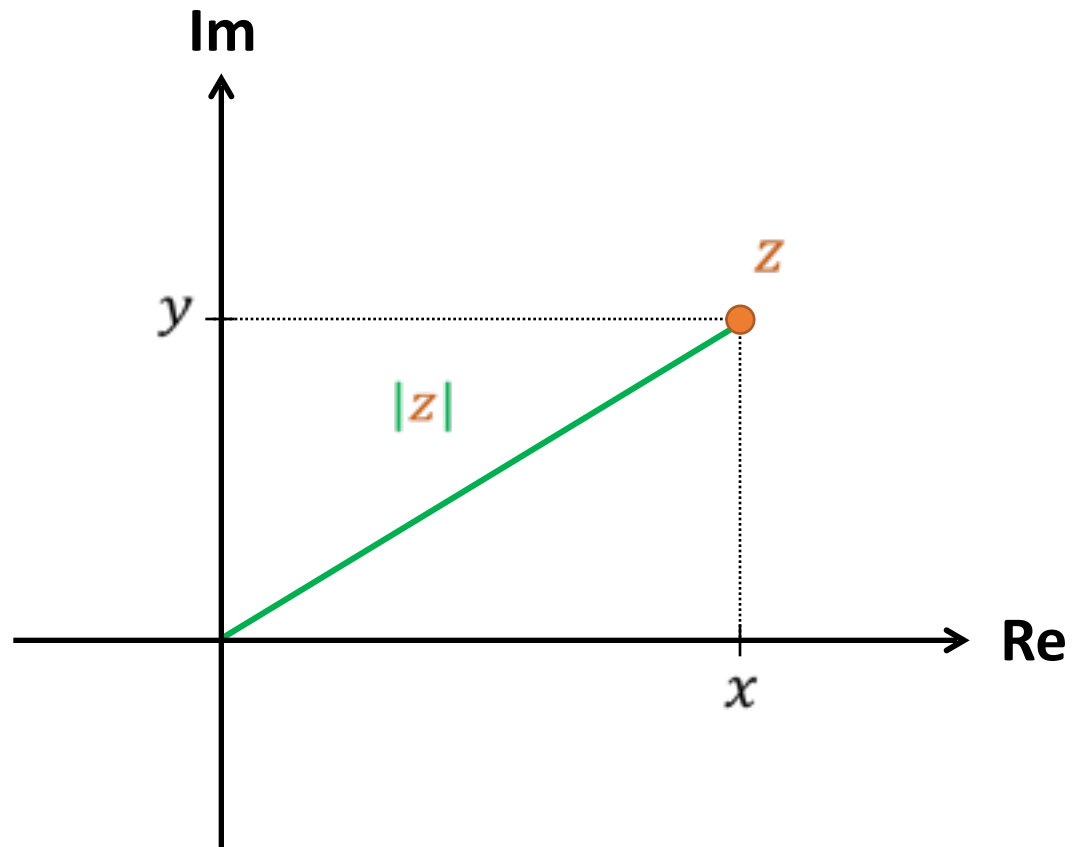
Complex Numbers – Basics



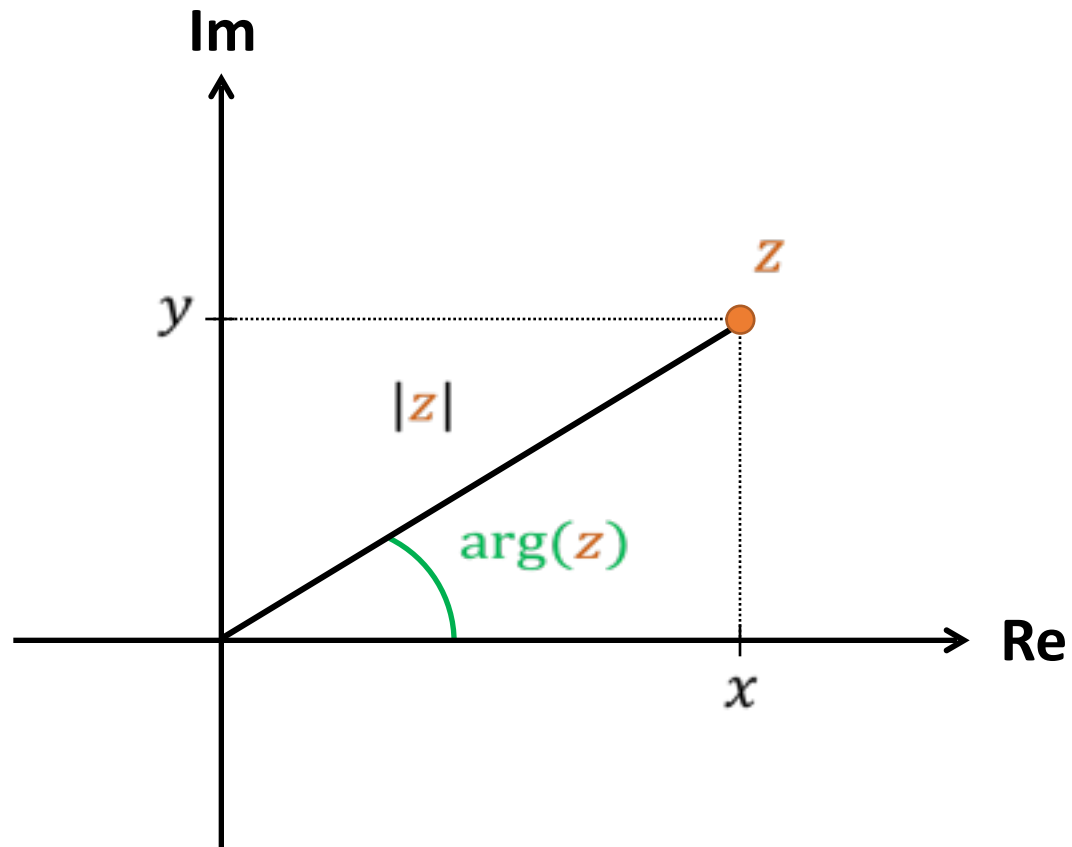
Complex Numbers – Basics



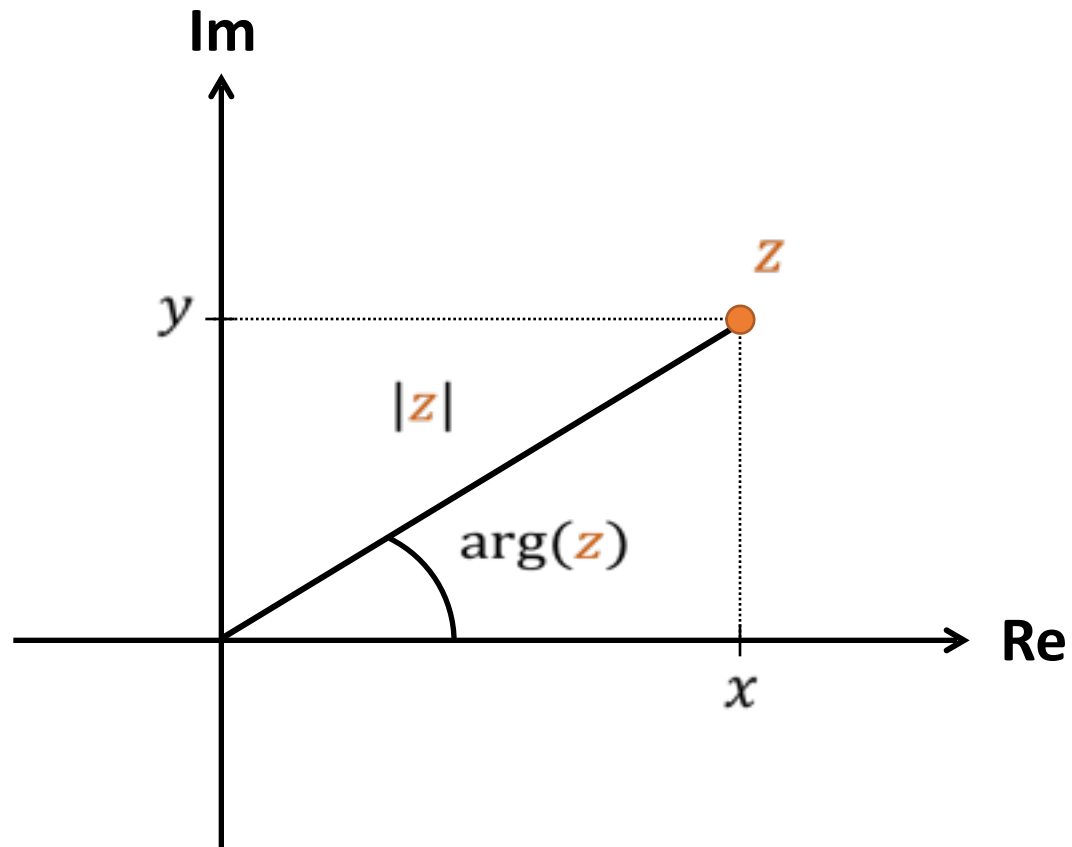
Complex Numbers – Basics



Complex Numbers – Basics



Complex Numbers – Basics



Classes - data members

- ◆ Define a class for complex numbers:
- ◆ Data members?
 - real part
 - imaginary part

Classes - data members

- ◆ Define a class for complex numbers:
- ◆ Data members?
 - real part
 - imaginary part

```
class complex {  
  
    private:  
        float real_;  
        float imag_;  
};
```

```
class complex {
```

```
private:  
    float real_;  
    float imag_;  
};
```

Classes - constructors

- ◆ Private members are not reachable from the outside
- ◆ *public* functions are used for read/write operations.
- ◆ Constructors: for initialization (public section of complex)


```
class complex {
```

```
private:  
    float real_;  
    float imag_;  
};
```

Classes - constructors

- ◆ Private members are not reachable from the outside
- ◆ *public* functions are used for read/write operations.
- ◆ Constructors: for initialization (public section of complex)

```
complex () : real_(0), imag_(0) {}
```

```
complex (const float& real) : real_(real), imag_(0) {}
```

```
complex (const float& real, const float& imag) : real_(real), imag_(imag) {}
```

```
class complex {
```

```
private:  
    float real_;  
    float imag_;  
};
```

Classes - constructors

- ◆ Private members are not reachable from the outside
- ◆ *public* functions are used for read/write operations.
- ◆ Constructors: for initialization (public section of complex)

```
complex () : real_(0), imag_(0) {}  
complex (const float& real) : real_(real), imag_(0) {}  
complex (const float& real, const float& imag) : real_(real), imag_(imag) {}
```

- ◆ When/how to use constructors?
- ◆ When declaring new variable! (in e.g. main)

Classes - constructors

- ◆ Private members are not reachable from the outside
- ◆ *public* functions are used for read/write operations.
- ◆ Constructors: for initialization (public section of complex)

```
complex () : real_(0), imag_(0) {}  
complex (const float& real) : real_(real), imag_(0) {}  
complex (const float& real, const float& imag) : real_(real), imag_(imag) {}
```

- ◆ When/how to use constructors?
- ◆ When declaring new variable! (in e.g. main)

```
complex x;  
complex y(3,4);  
complex z(4);    // constructor allowing conversion from float to complex
```

Classes - getters

- ◆ Used to retrieve the values of the private variables
- ◆ *Member functions*
- ◆ Defined in *public* section of the class

```
float get_real () const {return real_;}  
float get_imag () const {return imag_;}  
};
```

- ◆ `const` - prevents changes to `this` object.
- ◆ Member functions get the variable they operate on as an implicit argument

Getter functions - usage

```
class complex {
```

```
private:  
    float real_;  
    float imag_;  
};
```

```
float real = y.get_real();  
float const_imag = a.get_imag();
```

```
class complex {
```

```
private:  
    float real_;  
    float imag_;  
};
```

Classes - setters

- ◆ Same idea as getter functions - used to set the values of private members

```
void set_real (const float & real) {real_ = real;}  
void set_imag (const float & imag) {imag_ = imag;}
```

- ◆ Can no longer be const
 - ◆ why?

Classes - arithmetic operators

- ◆ Motivation: define basic arithmetic operations for complex numbers.

```
y += z;   y *= a;   x = y + z;   x = y * z;
```

- ◆ Before: two arguments.
- ◆ Now: operators can be class members
- ◆ Get a **this** object implicitly - only one argument is required.
- ◆ Again in the *public* part of the class definition
- ◆ Declaration for +=?

Classes - arithmetic operators

- ◆ In class declaration:

```
complex& operator+= (const complex& b);
```

- ◆ Out of class definition:

```
complex& complex::operator+= (const complex& b) {  
    real += b.real;  
    imag += b.imag;  
    return *this;  
}
```

- ◆ :: indicates to which class the operator belongs

Classes - stream operators

- ◆ Output the values of a complex number to a stream:

```
std::ostream& operator<< (std::ostream& o, const complex& z) {  
    o << "(" << z.get_real() << "," << z.get_imag() << " )";  
    return o;  
}
```

Agenda

- ◆ Homework
- ◆ Classes
- ◆ **Dynamically Allocated Memory**
- ◆ Dynamic Data Structures
- ◆ Dynamic Storage

Dynamically Allocated Memory

- ◆ Instruct the computer to give the requested number of memory blocks for a given data type
- ◆ Single memory block:

```
int* dyn_int = new int (3); // constructed with value 3
```

- ◆ Range of memory blocks:

```
int n = ...;  
int* dyn_int_range = new int[n];
```

Dynamically Allocated Memory

- ◆ The memory needs to be freed *explicitly* after usage.

```
delete dyn_int; // deconstruct dynamic variable  
dyn_int = 0;  
delete[] dyn_int_range; // deconstruct dynamic range  
dyn_int_range = 0;
```

Dynamically Allocated Memory

- ◆ Dynamically allocate storage to store n numbers and print them in reverse.

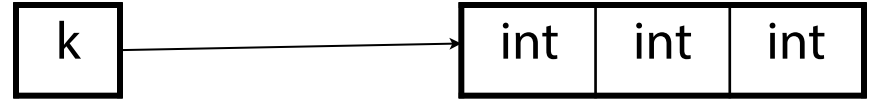
```
int i; std::cin >> i;    // length of input
int* mem = new int[i];
```

Dynamic Memory Allocation Dangers!

- `int * k = new int [3];`

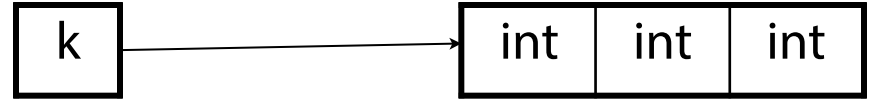
Dynamic Memory Allocation Dangers!

■ `int * k = new int [3];`



Dynamic Memory Allocation Dangers!

- `int * k = new int [3];`

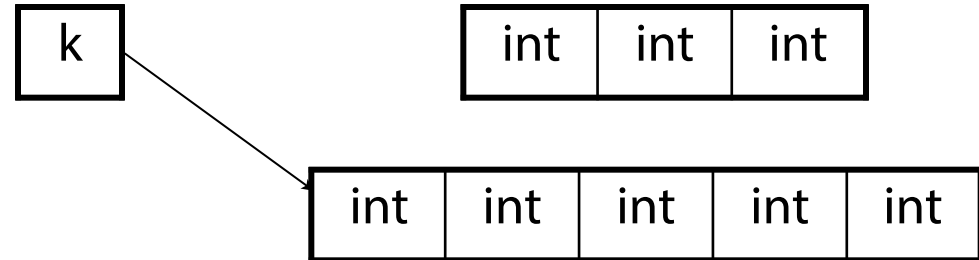


- `k = new int [5];`

Dynamic Memory Allocation Dangers!

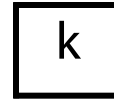
- `int * k = new int [3];`

- `k = new int [5];`



Dynamic Memory Allocation Dangers!

- `int * k = new int [3];`



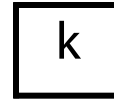
- `k = new int [5];` `//memory leak`



- `int * l = k;`

Dynamic Memory Allocation Dangers!

- `int * k = new int [3];`



- `k = new int [5];` `//memory leak`



- `int * l = k;`



Dynamic Memory Allocation Dangers!

- `int * k = new int [3];`



- `k = new int [5];` `//memory leak`



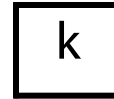
- `int * l = k;`



- `delete [] k;`

Dynamic Memory Allocation Dangers!

- `int * k = new int [3];`



- `k = new int [5];` `//memory leak`

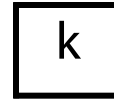
- `int * l = k;`



- `delete [] k;`

Dynamic Memory Allocation Dangers!

- `int * k = new int [3];`



- `k = new int [5];` `//memory leak`

- `int * l = k;`



- `delete [] k;` `//dangling pointer`

Agenda

- ◆ Homework
- ◆ Classes
- ◆ Dynamically Allocated Memory
- ◆ **Dynamic Data Structures**
- ◆ Dynamic Storage

Dynamic Data Structures

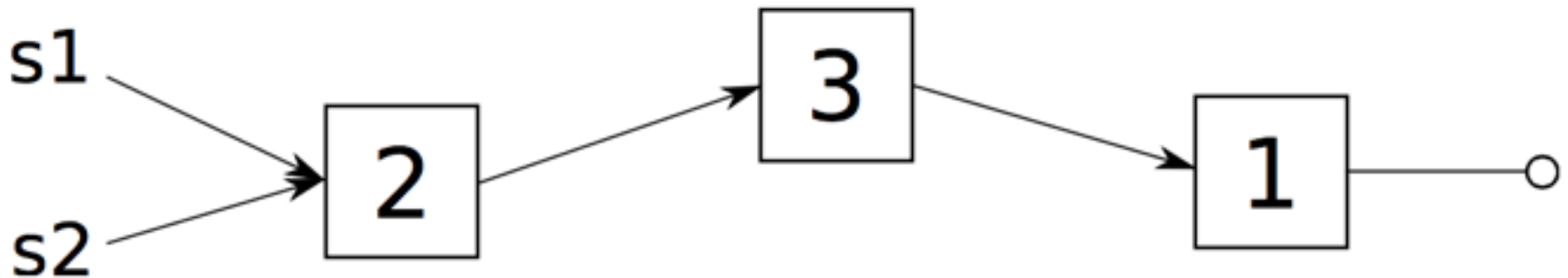
- ◆ Compiler creates a default copy constructor
- ◆ The default copy constructor will construct new a object and copy data members over.

```
ifmp::stack s1;  
s1.push(1);  
s1.push(3);  
s1.push(2);  
ifmp::stack s2 = s1;
```


Default copy constructor

```
ifmp::stack s1;  
s1.push(1);  
s1.push(3);  
s1.push(2);  
ifmp::stack s2 = s1;
```

Default copy constructor copies over the value of the top_node of s1 to s2:



Copying the entire data structure

- ◆ To copy the entire contents - implement the copy constructor and a recursive copy function:

```
stack::stack (const stack& s) : top_node (0) {  
    copy(s.top_node, top_node);  
}
```

```
void stack::copy(const linked_list_node* from, linked_list_node*& to) {  
    assert (to == 0);  
    if (from != 0) {  
        to = new linked_list_node (from->key);  
        copy (from->next, to->next);  
    }  
}
```

Copy constructors

- ◆ Copy constructors take a **reference** as an argument.
- ◆ What would happen if they took an argument by value?

```
stack::stack(const stack s)
```

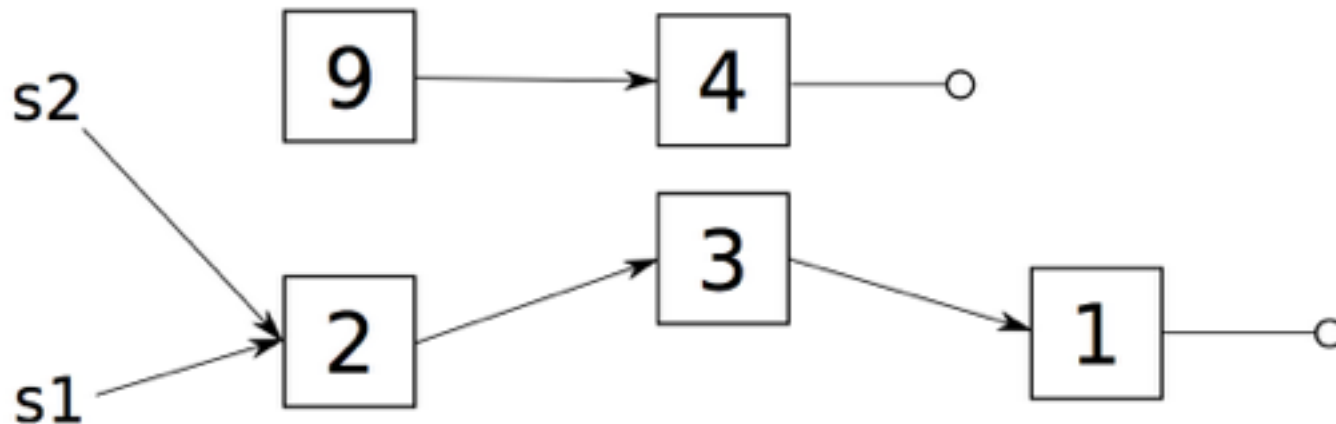
- ◆ Calling by value means using the copy constructor - infinite recursion.

Copy constructors

- ◆ What happens in this case?

```
ifmp::stack s2;  
s.push(4);  
s.push(9);
```

```
s2 = s1; // s1 as before
```



Copy constructors

- ◆ How can we fix this?
- ◆ Clear the old data before copying new elements.

```
void stack::clear(linked_list_node* from) {  
    if (from != 0) {  
        clear (from->next);  
        delete from;  
    }  
}
```

- ◆ Is this correct for all cases?

Copy constructors

- ◆ Need to check for self-assignment in `operator=` by comparing the top node addresses:

```
top_node != s.top_node
```

```
stack& stack::operator= (const stack& s) {  
    if (top_node != s.top_node) { // test for self-assignment  
        clear (top_node);  
        top_node = 0; // fix dangling pointer  
        copy (s.top_node, top_node);  
    }  
    return *this;  
}
```

Destructors

- ◆ Dynamically allocated memory needs to be freed when the class is destructed.

```
stack::~~stack() { clear(top_node); }
```

Agenda

- ◆ Homework
- ◆ Classes
- ◆ Dynamically Allocated Memory
- ◆ Dynamic Data Structures
- ◆ **Dynamic Storage**

Dynamic Storage Exercise

Dynamic Storage Exercise

```
int i;  
while (std::cin >> i) ...
```

reads inputs as long as there are more available.

Write a code snippet which reads inputs as described above, and which then stores these inputs in an array. For this exercise you are not allowed to use the Standard Library (i.e. **no** `std::vector`).

To achieve this you will have to use `new[]` and `delete[]`.

Dynamic Storage Solution

- Idea:



Dynamic Storage Solution

- Idea:
 1. Allocate some range (using `new []`)



Dynamic Storage Solution

- Idea:
 1. Allocate some range (using `new []`)
 2. As soon as range full,



Dynamic Storage Solution

- Idea:
 1. Allocate some range (using `new []`)
 2. As soon as range full,



Dynamic Storage Solution

- Idea:
 1. Allocate some range (using `new []`)
 2. As soon as range full,



Dynamic Storage Solution

- Idea:
 1. Allocate some range (using `new []`)
 2. As soon as range full, allocate larger range (using `new []`)



Dynamic Storage Solution

- Idea:
 1. Allocate some range (using `new []`)
 2. As soon as range full, allocate larger range (using `new []`)
 3. Copy over initial range



Dynamic Storage Solution

- Idea:
 1. Allocate some range (using `new []`)
 2. As soon as range full, allocate larger range (using `new []`)
 3. Copy over initial range



Dynamic Storage Solution

- Idea:
 1. Allocate some range (using `new []`)
 2. As soon as range full, allocate larger range (using `new []`)
 3. Copy over initial range



Dynamic Storage Solution

- Idea:
 1. Allocate some range (using `new []`)
 2. As soon as range full, allocate larger range (using `new []`)
 3. Copy over initial range



Dynamic Storage Solution

- Idea:
 1. Allocate some range (using `new []`)
 2. As soon as range full, allocate larger range (using `new []`)
 3. Copy over initial range
 4. Delete initial range (using `delete []`)



Dynamic Storage Solution

- Idea:
 1. Allocate some range (using `new []`)
 2. As soon as range full, allocate larger range (using `new []`)
 3. Copy over initial range
 4. Delete initial range (using `delete []`)
 5. Go back to 2. with newly generated memory



Dynamic Storage Solution

- Idea:
 1. Allocate some range (using `new []`)
 2. As soon as range full, allocate larger range (using `new []`)
 3. Copy over initial range
 4. Delete initial range (using `delete []`)
 5. Go back to 2. with newly generated memory



Dynamic Storage Solution

- Idea:
 1. Allocate some range (using `new []`)
 2. As soon as range full, allocate larger range (using `new []`)
 3. Copy over initial range
 4. Delete initial range (using `delete []`)
 5. Go back to 2. with newly generated memory

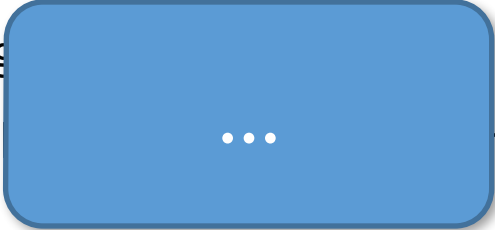


Dynamic Storage Solution

- Idea:
 1. Allocate some range (using `new []`)
 2. As soon as range full, allocate larger range (using `new []`)
 3. Copy over initial range
 4. Delete initial range (using `delete []`)
 5. Go back to 2. with newly generated memory



Dynamic Storage Solution

- Idea:
 1. Allocate some range (using `new []`)
 2. As soon as range full, allocate larger range (using `new []`)
 3. Copy over initial range
 4. Delete initial range
 5. Go back to 2. with  memory



Dynamic Storage Solution

```
int n = 1; // current array size
int k = 0; // number of elements read so far

// dynamically allocate array
int* a = new int[n]; // this time, n is NOT a constant

// read into the array
while (std::cin >> a[k]) {
    if (++k == n) {
        // next element wouldn't fit; replace the array a by
        // a new one of twice the size
        int* b = new int[n*=2]; // get pointer to new array
        for (int i=0; i<k; ++i) // copy old array to new one
            b[i] = a[i];
        delete[] a; // delete old array
        a = b; // let a point to new array
    }
}

...

delete[] a; // don't forget to delete after use
```

New Range - How Much Larger?

Dynamic Storage Solution

- "Much" larger?
 - Pro: ranges less often full → copy less often
 - Con: larger memory consumption
- Important: Larger by a **factor**, not by a **constant**...
 - $\text{length_n} = \text{length_o} * 2$
 $\text{length_n} = \text{length_o} + 2$

Dynamic Storage Solution

- Larger by: a) factor 2 b) constant 2

elements	Case a)	Case b)

Dynamic Storage Solution

- Larger by: a) factor 2 b) constant 2

elements	Case a)	Case b)
	2	2
	4	4
	4	4
	8	6
	8	6
	8	8
	8	8
	16	10
	16	10
	16	12
	16	12
	16	14
	16	14
	16	16
	16	16
	32	18



Dynamic Storage Solution

- Larger by: a) factor 2 b) constant 2

elements	Case a)	Case b)
	2	2
	4	4
	4	4
	8	6
	8	6
	8	8
	8	8
	16	10
	16	10
	16	12
	16	12
	16	14
	16	14
	16	16
	16	16
	32	18

← arbitr. chosen

Case a):

Significantly
fewer resizings.

Dynamic Storage Solution

- Larger by: a) factor 2 b) constant 2

elements	Case a)	Case b)
	2	2
	4	4
	4	4
	8	6
	8	6
	8	8
	8	8
	16	10
	16	10
	16	12
	16	12
	16	14
	16	14
	16	16
	16	16
	32	18

← arbitr. chosen

Each resizing
means:

Copy **WHOLE**
array.

Case a):

Significantly
fewer resizings.

Dynamic Storage Solution

- Larger by: a) factor 2 b) c

elements	Case a)	Case b)
	2	2
	4	4
	8	8
	16	16
	32	32
	64	64
	128	128
	256	256
	512	512
	1024	1024
	2048	2048
	4096	4096
	8192	8192
	16384	16384
	32768	32768
	65536	65536
	131072	131072
	262144	262144
	524288	524288
	1048576	1048576
	2097152	2097152
	4194304	4194304
	8388608	8388608
	16777216	16777216
	33554432	33554432
	67108864	67108864
	134217728	134217728
	268435456	268435456
	536870912	536870912
	1073741824	1073741824
	2147483648	2147483648
	4294967296	4294967296
	8589934592	8589934592
	17179869184	17179869184
	34359738368	34359738368
	68719476736	68719476736
	137438953472	137438953472
	274877906944	274877906944
	549755813888	549755813888
	1099511627776	1099511627776
	2199023255552	2199023255552
	4398046511104	4398046511104
	8796093022208	8796093022208
	17592186044416	17592186044416
	35184372088832	35184372088832
	70368744177664	70368744177664
	140737488355328	140737488355328
	281474976710656	281474976710656
	562949953421312	562949953421312
	1125899906842624	1125899906842624
	2251799813685248	2251799813685248
	4503599627370496	4503599627370496
	9007199254740992	9007199254740992
	18014398509481984	18014398509481984
	36028797018963968	36028797018963968
	72057594037927936	72057594037927936
	144115188075855872	144115188075855872
	288230376151711744	288230376151711744
	576460752303423488	576460752303423488
	1152921504606846976	1152921504606846976
	2305843009213693952	2305843009213693952
	4611686018427387904	4611686018427387904
	9223372036854775808	9223372036854775808
	18446744073709551616	18446744073709551616
	36893488147419103232	36893488147419103232
	73786976294838206464	73786976294838206464
	147573952589676412928	147573952589676412928
	295147905179352825856	295147905179352825856
	590295810358705651712	590295810358705651712
	1180591620717411303424	1180591620717411303424
	2361183241434822606848	2361183241434822606848
	4722366482869645213696	4722366482869645213696
	9444732965739290427392	9444732965739290427392
	18889465931478580854784	18889465931478580854784
	37778931862957161709568	37778931862957161709568
	75557863725914323419136	75557863725914323419136
	151115727451828646838272	151115727451828646838272
	302231454903657293676544	302231454903657293676544
	604462909807314587353088	604462909807314587353088
	1208925819614629174706176	1208925819614629174706176
	2417851639229258349412352	2417851639229258349412352
	4835703278458516698824704	4835703278458516698824704
	9671406556917033397649408	9671406556917033397649408
	19342813113834066795298816	19342813113834066795298816
	38685626227668133590597632	38685626227668133590597632
	77371252455336267181195264	77371252455336267181195264
	154742504910672534362390528	154742504910672534362390528
	309485009821345068724781056	309485009821345068724781056
	618970019642690137449562112	618970019642690137449562112
	1237940039285380274899124224	1237940039285380274899124224
	2475880078570760549798248448	2475880078570760549798248448
	4951760157141521099596496896	4951760157141521099596496896
	9903520314283042199192993792	9903520314283042199192993792
	19807040628566084398385987584	19807040628566084398385987584
	39614081257132168796771975168	39614081257132168796771975168
	79228162514264337593543950336	79228162514264337593543950336
	158456325028528675187087900672	158456325028528675187087900672
	316912650057057350374175801344	316912650057057350374175801344
	633825300114114700748351602688	633825300114114700748351602688
	1267650600228229401496703205376	1267650600228229401496703205376
	2535301200456458802993406410752	2535301200456458802993406410752
	5070602400912917605986812821504	5070602400912917605986812821504
	10141204801825835211973625643008	10141204801825835211973625643008
	20282409603651670423947251286016	20282409603651670423947251286016
	40564819207303340847894502572032	40564819207303340847894502572032
	81129638414606681695789005144064	81129638414606681695789005144064
	162259276829213363391578010288128	162259276829213363391578010288128
	324518553658426726783156020576256	324518553658426726783156020576256
	649037107316853453566312041152512	649037107316853453566312041152512
	1298074214633706907132624082305024	1298074214633706907132624082305024
	2596148429267413814265248164610048	2596148429267413814265248164610048
	5192296858534827628530496329220096	5192296858534827628530496329220096
	10384593717069655257060992658440192	10384593717069655257060992658440192
	20769187434139310514121985316880384	20769187434139310514121985316880384
	41538374868278621028243970633760768	41538374868278621028243970633760768
	83076749736557242056487941267521536	83076749736557242056487941267521536
	166153499473114484112975882535043072	166153499473114484112975882535043072
	332306998946228968225951765070086144	332306998946228968225951765070086144
	664613997892457936451903530140172288	664613997892457936451903530140172288
	1329227995784915872903807060280344576	1329227995784915872903807060280344576
	2658455991569831745807614120560689152	2658455991569831745807614120560689152
	5316911983139663491615228241121378304	5316911983139663491615228241121378304
	10633823966279326983230456482242756608	10633823966279326983230456482242756608
	21267647932558653966460912964485513216	21267647932558653966460912964485513216
	42535295865117307932921825928971026432	42535295865117307932921825928971026432
	85070591730234615865843651857942052864	85070591730234615865843651857942052864
	170141183460469231731687303715884105728	170141183460469231731687303715884105728
	340282366920938463463374607431768211456	340282366920938463463374607431768211456
	680564733841876926926749214863536422912	680564733841876926926749214863536422912
	1361129467683753853853498429727072845824	1361129467683753853853498429727072845824
	2722258935367507707706996859454145691648	2722258935367507707706996859454145691648
	5444517870735015415413993718908291383296	5444517870735015415413993718908291383296
	10889035741470030830827987437816582766592	10889035741470030830827987437816582766592
	21778071482940061661655974875633165533184	21778071482940061661655974875633165533184
	43556142965880123323311949751266331066368	43556142965880123323311949751266331066368
	87112285931760246646623899502532662132736	87112285931760246646623899502532662132736
	174224571863520493293247799005065324265472	174224571863520493293247799005065324265472
	348449143727040986586495598010130648530944	348449143727040986586495598010130648530944
	696898287454081973172991196020261297061888	696898287454081973172991196020261297061888
	1393796574908163946345982392040522594123776	1393796574908163946345982392040522594123776
	2787593149816327892691964784081045188247552	2787593149816327892691964784081045188247552
	5575186299632655785383929568162090376495104	5575186299632655785383929568162090376495104
	11150372599265311570767859136324180752990208	11150372599265311570767859136324180752990208
	22300745198530623141535718272648361505980416	22300745198530623141535718272648361505980416
	44601490397061246283071436545296723011960832	44601490397061246283071436545296723011960832
	89202980794122492566142873090593446023921664	89202980794122492566142873090593446023921664
	178405961588244985132285746181186892047843328	178405961588244985132285746181186892047843328
	356811923176489970264571492362373784095686656	356811923176489970264571492362373784095686656
	713623846352979940529142984724747568191373312	713623846352979940529142984724747568191373312
	1427247692705959881058285969449495136382746624	1427247692705959881058285969449495136382746624
	2854495385411919762116571938898990272765493248	2854495385411919762116571938898990272765493248
	5708990770823839524233143877797980545530986496	5708990770823839524233143877797980545530986496
	11417981541647679048466287755595961091061972992	11417981541647679048466287755595961091061972992
	22835963083295358096932575511191922182123945984	22835963083295358096932575511191922182123945984
	45671926166590716193865151022383844364247891968	45671926166590716193865151022383844364247891968
	91343852333181432387730302044767688728495783936	91343852333181432387730302044767688728495783936
	182687704666362864775460604089535377456991567872	182687704666362864775460604089535377456991567872
	365375409332725729550921208179070754913983135744	365375409332725729550921208179070754913983135744
	730750818665451459101842416358141509827966271488	730750818665451459101842416358141509827966271488
	1461501637330902918203684832716283019655932542976	1461501637330902918203684832716283019655932542976
	2923003274661805836407369665432566039311865085952	2923003274661805836407369665432566039311865085952
	5846006549323611672814739330865132078623730171904	5846006549323611672814739330865132078623730171904
	11692013098647223345629478661730264157247460343808	11692013098647223345629478661730264157247460343808
	23384026197294446691258957323460528314494920687616	23384026197294446691258957323460528314494920687616
	46768052394588893382517914646921056628989841375232	46768052394588893382517914646921056628989841375232
	93536104789177786765035829293842113257979682750464	93536104789177786765035829293842113257979682750464
	187072209578355573530071658587684226515959365500928	187072209578355573530071658587684226515959365500928
	374144419156711147060143317175368453031918731001856	374144419156711147060143317175368453031918731001856
	748288838313422294120286634350736906063837462003712	748288838313422294120286634350736906063837462003712
	1496577676626844588240573268701473812127674924007424	1496577676626844588240573268701473812127674924007424
	2993155353253689176481146537402947624255349848014848	2993155353253689176481146537402947624255349848014848
	5986310706507378352962293074805895248510699696029696	5986310706507378352962293074805895248510699696029696
	119726214	

Vectors

Vectors

- Vectors can grow!

```
std::vector<int> vec (2,0); // 0 0
vec.push_back(7);          // 0 0 7
vec.push_back(2);          // 0 0 7 2
vec.push_back(6);          // 0 0 7 2 6
```

- This works as discussed before!

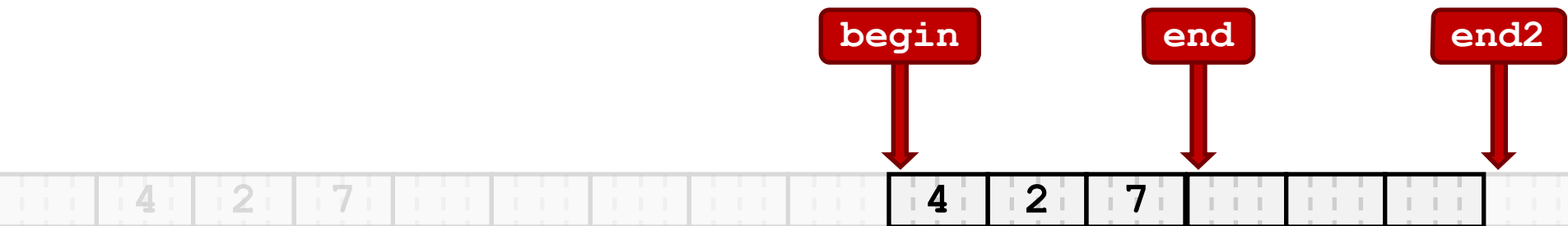
Dynamic Storage in Vectors

- Vectors store 3 pointers:

begin: begin of memory

end: end of *user-accessible* part

end2: end of allocated part



Dynamic Storage in Vectors

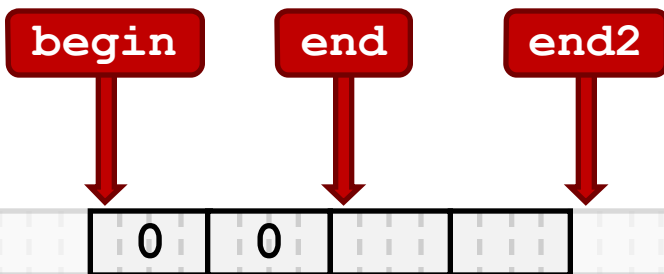
- Example:

```
ifmp::vector<int> vec (2,0); // 0 0  
vec.push_back(7);           // 0 0 7  
vec.push_back(2);           // 0 0 7 2  
vec.push_back(6);           // 0 0 7 2 6
```

Dynamic Storage in Vectors

- Example:

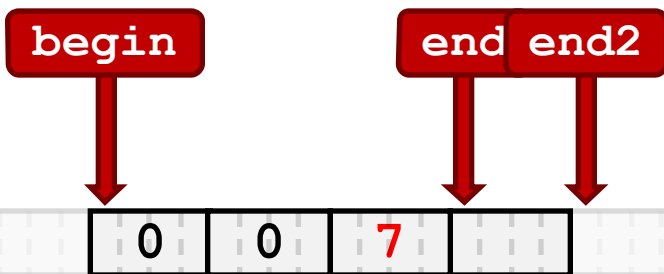
```
ifmp::vector<int> vec (2,0); // 0 0  
vec.push_back(7);           // 0 0 7  
vec.push_back(2);           // 0 0 7 2  
vec.push_back(6);           // 0 0 7 2 6
```



Dynamic Storage in Vectors

- Example:

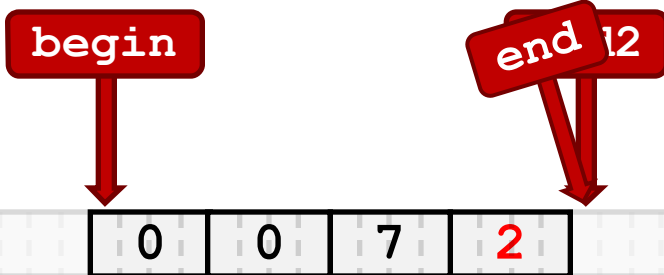
```
ifmp::vector<int> vec (2,0); // 0 0  
vec.push_back(7);           // 0 0 7  
vec.push_back(2);           // 0 0 7 2  
vec.push_back(6);           // 0 0 7 2 6
```



Dynamic Storage in Vectors

- Example:

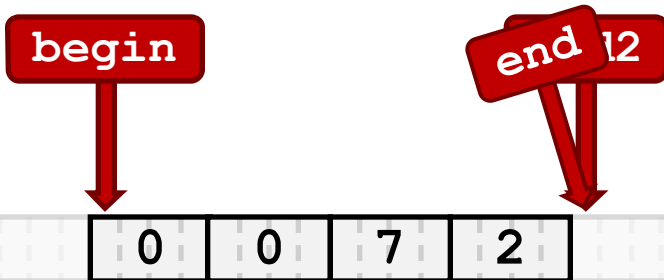
```
ifmp::vector<int> vec (2,0); // 0 0  
vec.push_back(7);           // 0 0 7  
vec.push_back(2);           // 0 0 7 2  
vec.push_back(6);           // 0 0 7 2 6
```



Dynamic Storage in Vectors

- Example:

```
ifmp::vector<int> vec (2,0); // 0 0  
vec.push_back(7);           // 0 0 7  
vec.push_back(2);           // 0 0 7 2  
vec.push_back(6);           // 0 0 7 2 6
```



Dynamic Storage in Vectors

- Example:

```
ifmp::vector<int> vec;  
vec.push_back(0);  
vec.push_back(0);  
vec.push_back(7);  
vec.push_back(2);
```

```
// 0 0  
0 0 7  
0 0 7 2  
0 7 2 6
```

Space full
Now:
copy range

begin

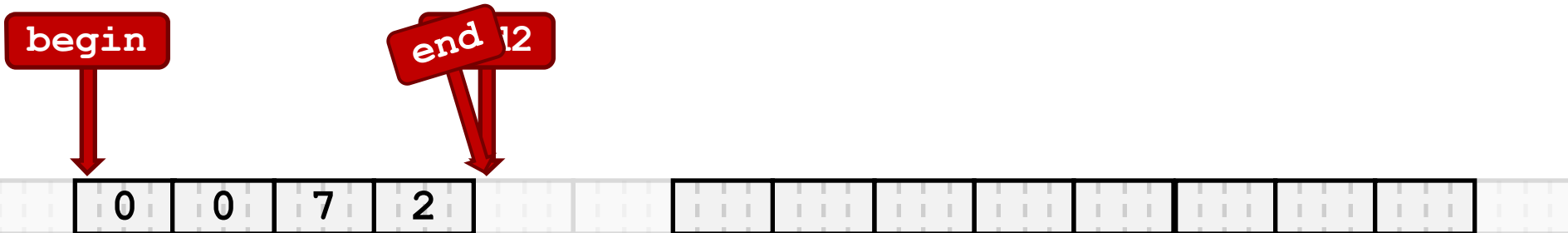
end 12

0 0 7 2

Dynamic Storage in Vectors

- Example:

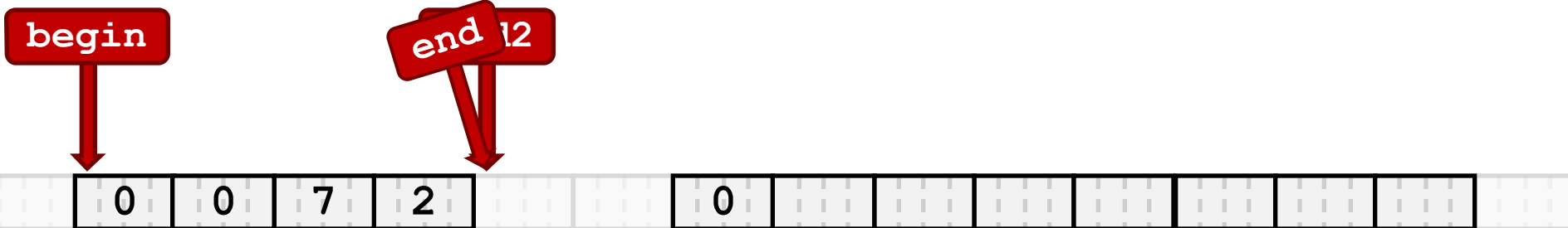
```
ifmp::vector<int> vec (2,0); // 0 0  
vec.push_back(7);           // 0 0 7  
vec.push_back(2);           // 0 0 7 2  
vec.push_back(6);           // 0 0 7 2 6
```



Dynamic Storage in Vectors

- Example:

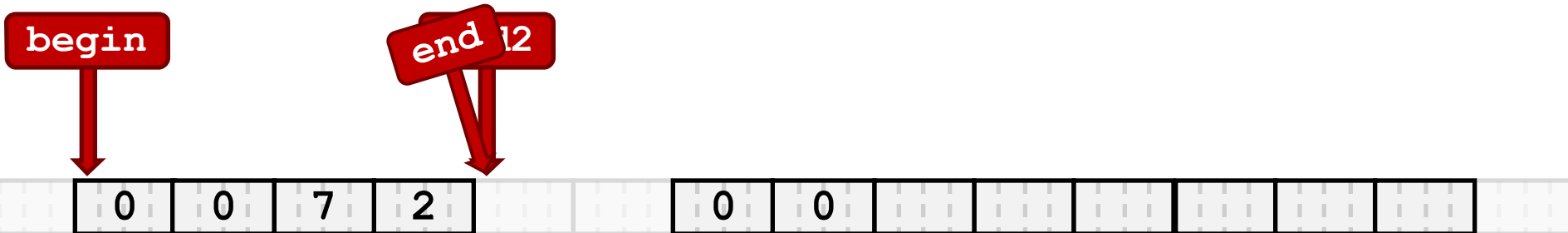
```
ifmp::vector<int> vec (2,0); // 0 0  
vec.push_back(7);           // 0 0 7  
vec.push_back(2);           // 0 0 7 2  
vec.push_back(6);           // 0 0 7 2 6
```



Dynamic Storage in Vectors

- Example:

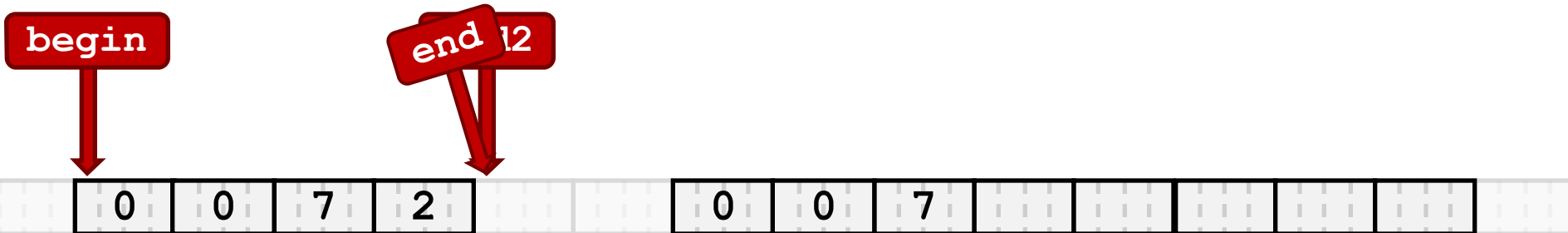
```
ifmp::vector<int> vec (2,0); // 0 0  
vec.push_back(7);           // 0 0 7  
vec.push_back(2);           // 0 0 7 2  
vec.push_back(6);           // 0 0 7 2 6
```



Dynamic Storage in Vectors

- Example:

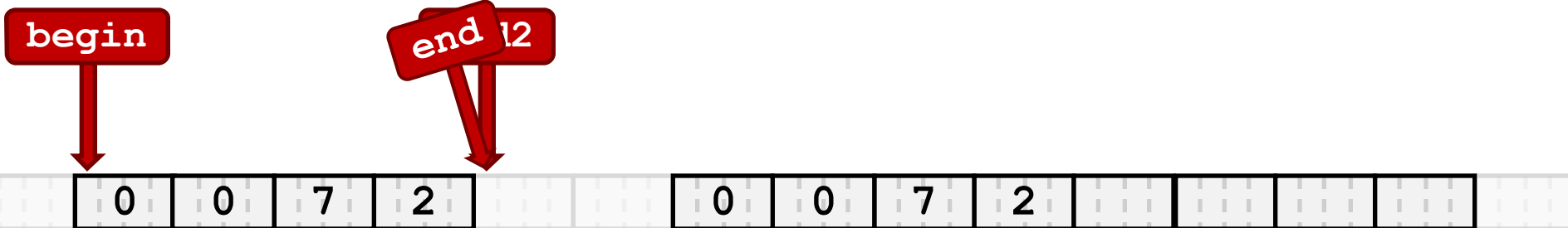
```
ifmp::vector<int> vec (2,0); // 0 0  
vec.push_back(7);           // 0 0 7  
vec.push_back(2);           // 0 0 7 2  
vec.push_back(6);           // 0 0 7 2 6
```



Dynamic Storage in Vectors

- Example:

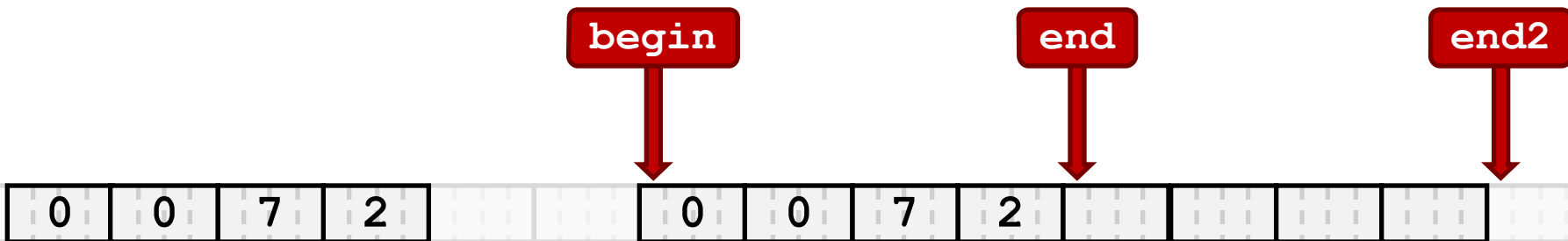
```
ifmp::vector<int> vec (2,0); // 0 0  
vec.push_back(7);           // 0 0 7  
vec.push_back(2);           // 0 0 7 2  
vec.push_back(6);           // 0 0 7 2 6
```



Dynamic Storage in Vectors

- Example:

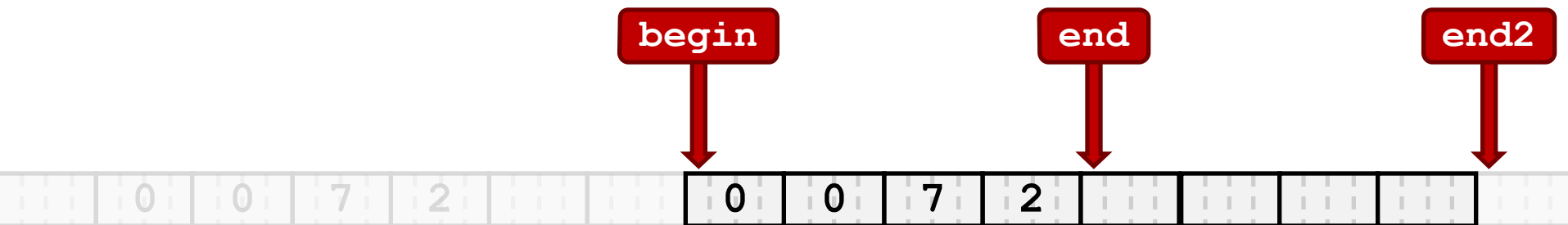
```
ifmp::vector<int> vec (2,0); // 0 0  
vec.push_back(7);           // 0 0 7  
vec.push_back(2);           // 0 0 7 2  
vec.push_back(6);           // 0 0 7 2 6
```



Dynamic Storage in Vectors

- Example:

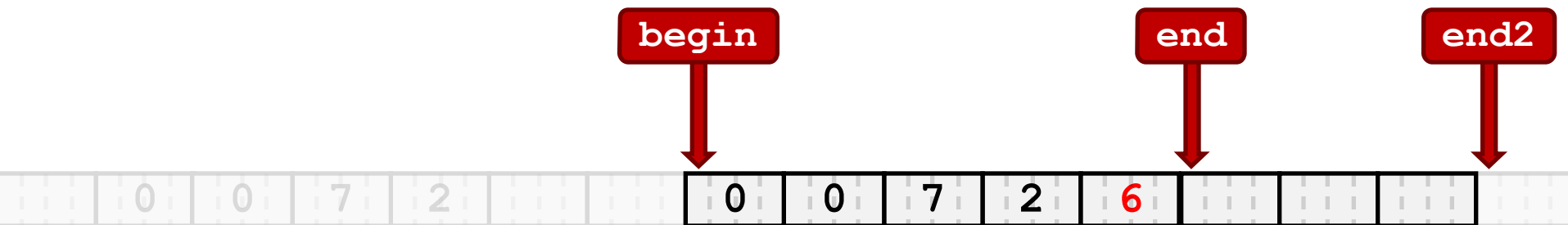
```
ifmp::vector<int> vec (2,0); // 0 0  
vec.push_back(7);           // 0 0 7  
vec.push_back(2);           // 0 0 7 2  
vec.push_back(6);           // 0 0 7 2 6
```



Dynamic Storage in Vectors

- Example:

```
ifmp::vector<int> vec (2,0); // 0 0
vec.push_back(7);           // 0 0 7
vec.push_back(2);           // 0 0 7 2
vec.push_back(6);           // 0 0 7 2 6
```



Dynamic Storage in Vectors

- Exercise sheet 12: implement your own vector type.
- Important:
 - **In constructor** Set initial range
 - **In copy-constructor** Don't copy just pointers;
i.e. copy the ranges behind them
 - **In operator=** Like copy-constructor, in addition:
 - i) prevent self-assignments
 - ii) don't forget to delete old range