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:

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

Complex Numbers – Basics

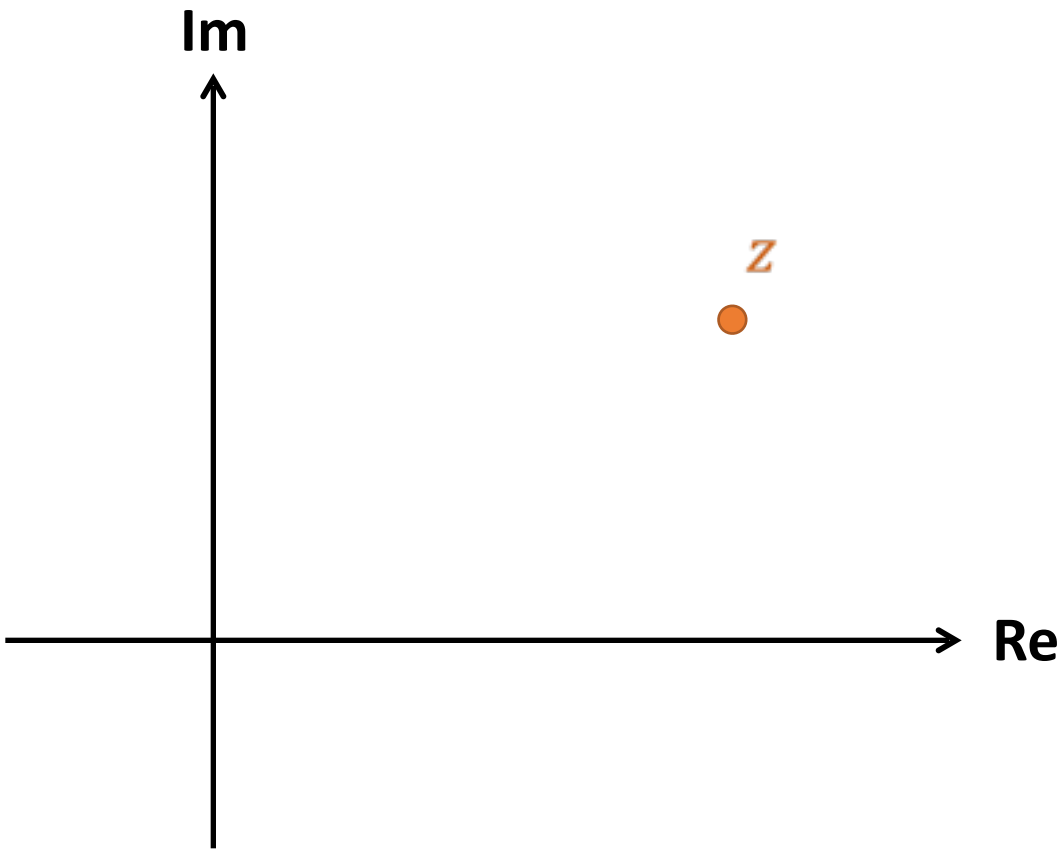
- Imaginary unit i:

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

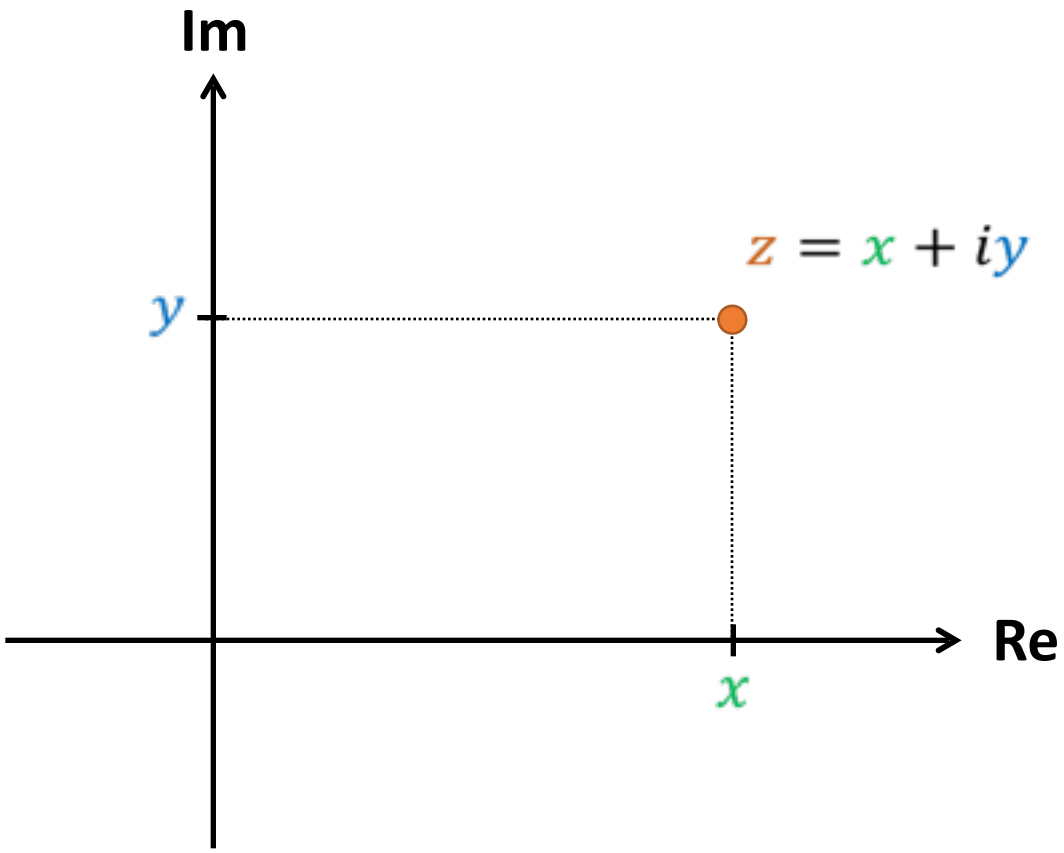
- $z \in \mathbb{C}$ can be represented as

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

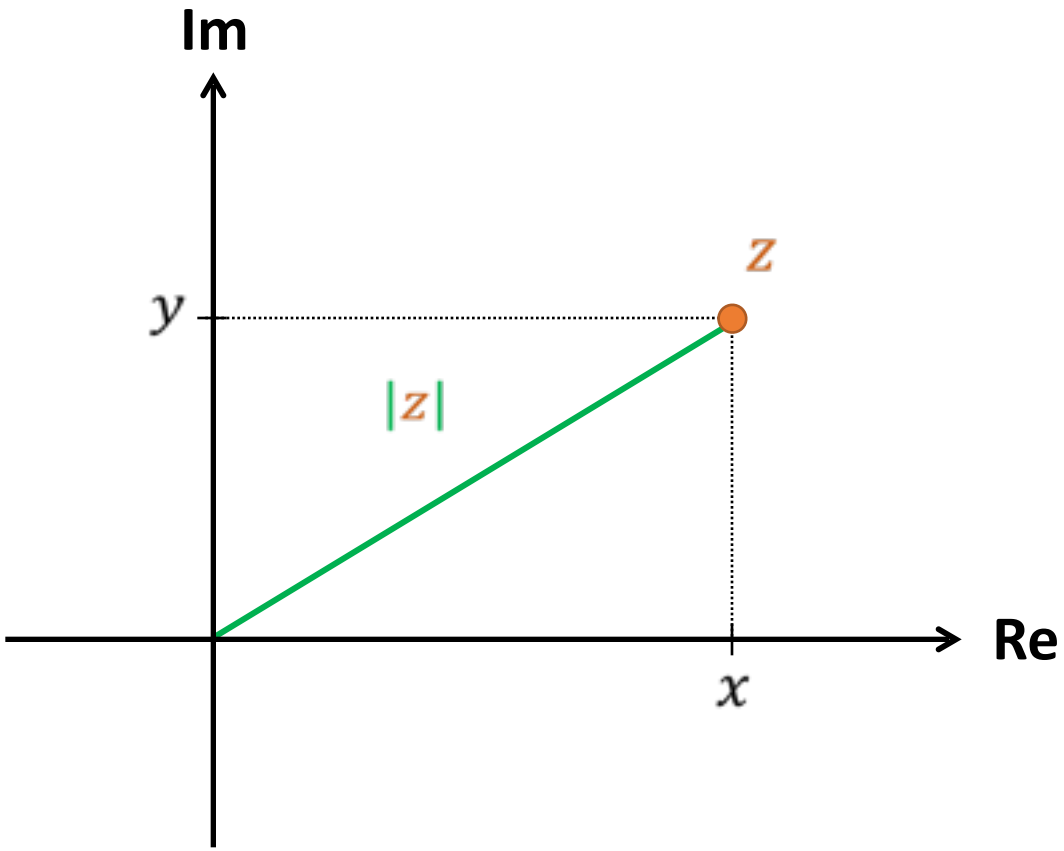
Complex Numbers – Basics



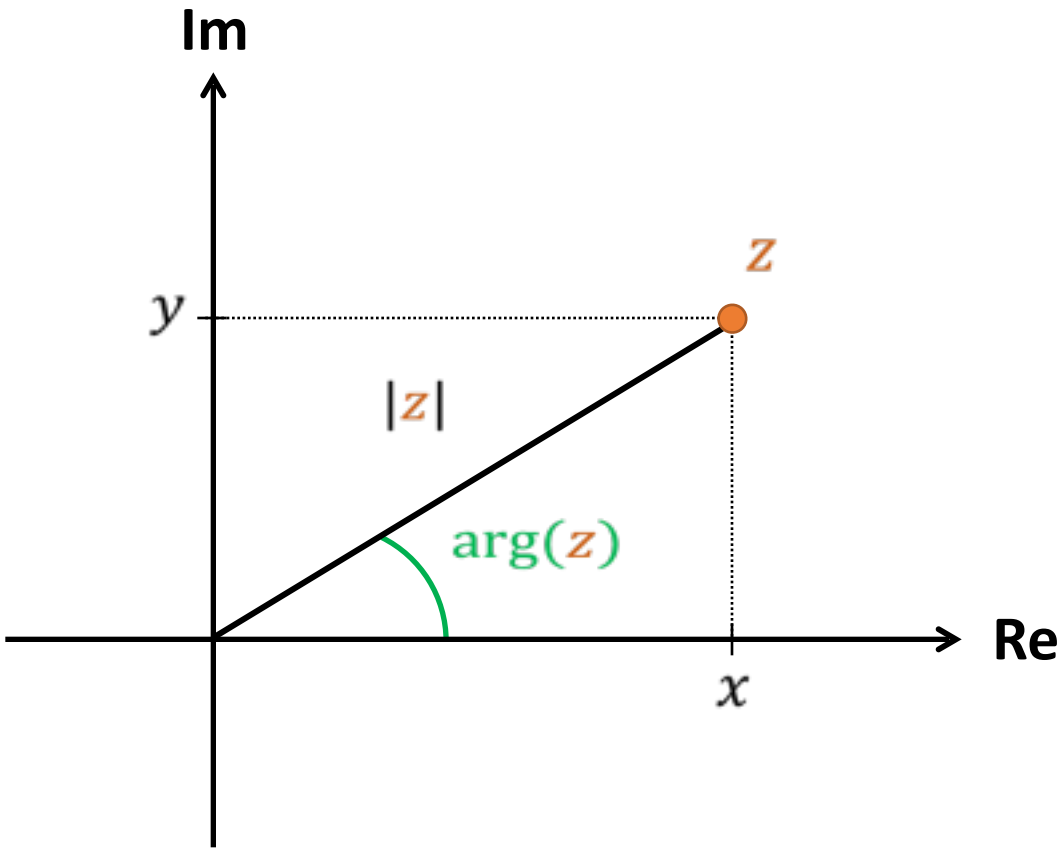
Complex Numbers – Basics



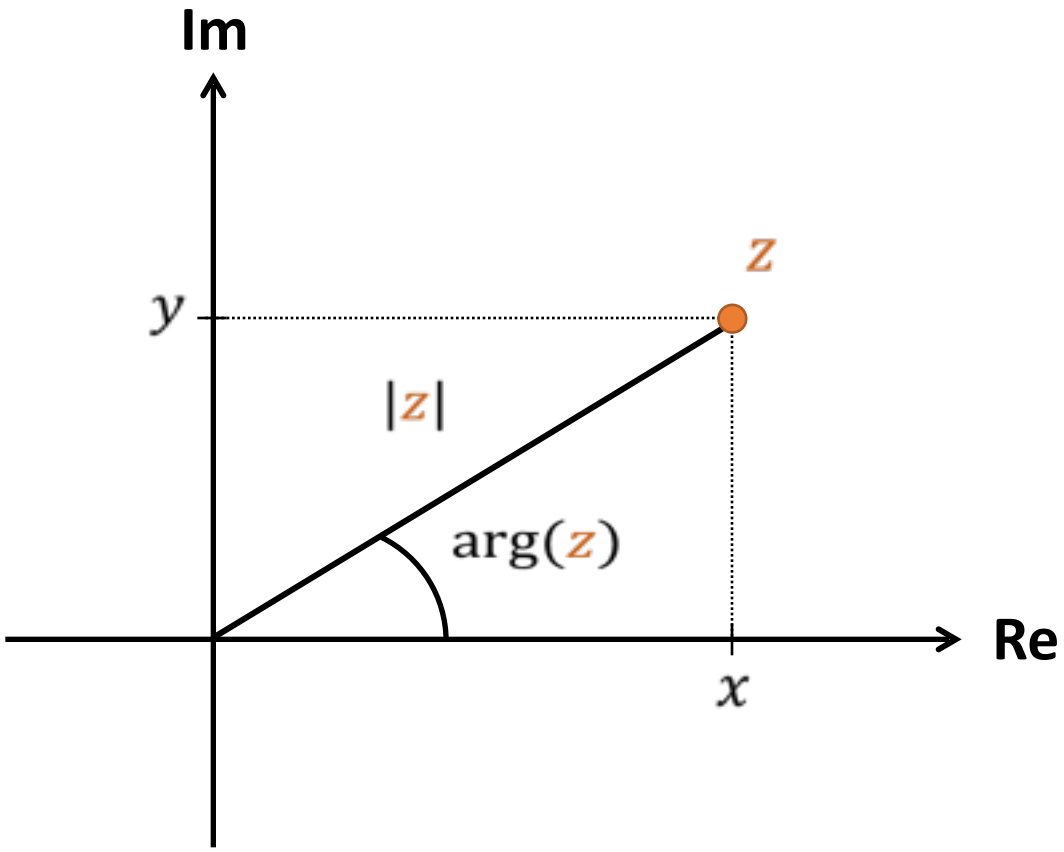
Complex Numbers – Basics



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_;
};
```

Classes - constructors

```
class complex {

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

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

Classes - constructors

```
class complex {

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

- 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) {}
```

Classes - constructors

```
class complex {

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

- 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

```
class complex {

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

- 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
```

```
class complex {

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

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();
```

Classes - setters

```
class complex {

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

- 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?

```
class complex {

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

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

```
class complex {

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

- **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

```
class complex {

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

- 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.

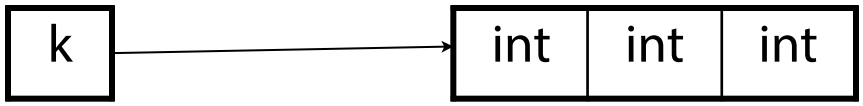
```
int i; std::cin >> i;   // length of input
int* mem = new int[i];
```

Dynamic Memory Allocation Dangers!

- int * k = new int [3];

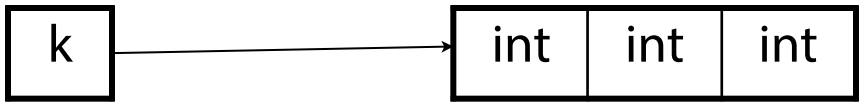
Dynamic Memory Allocation Dangers!

- int * k = new int [3];



Dynamic Memory Allocation Dangers!

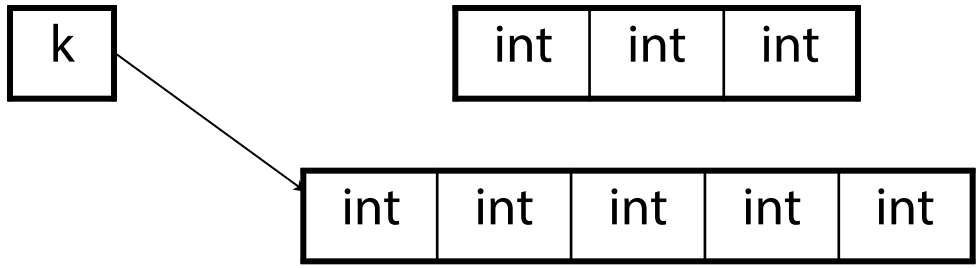
- int * k = new int [3];



- k = new int [5];

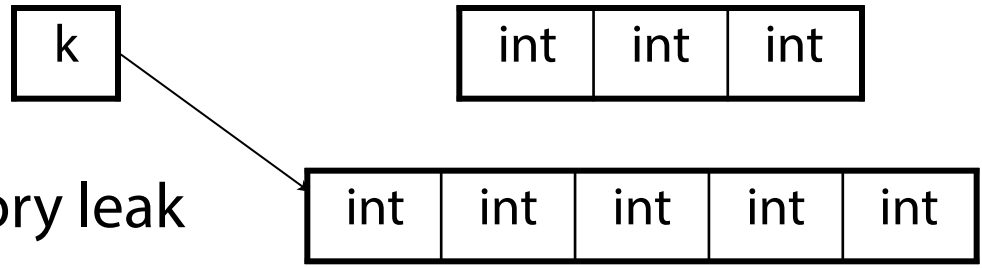
Dynamic Memory Allocation Dangers!

- int * k = new int [3];
- k = new int [5];



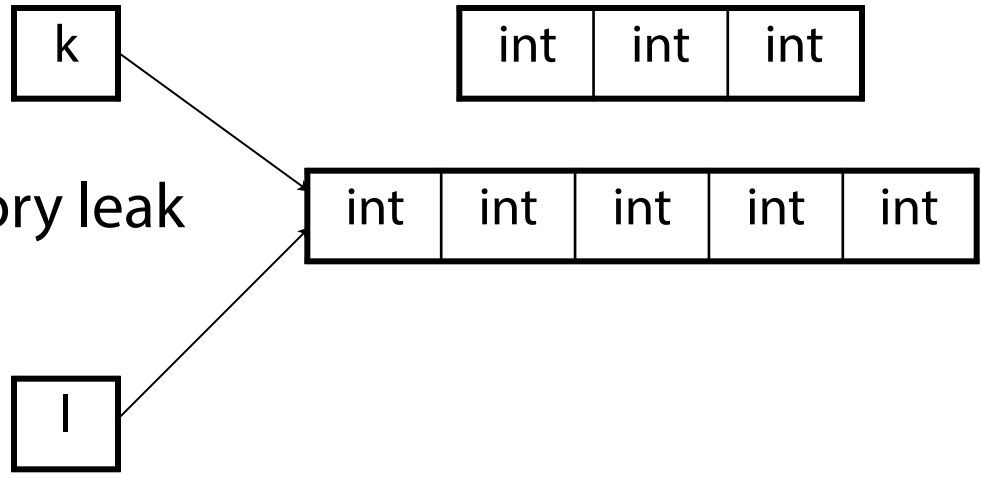
Dynamic Memory Allocation Dangers!

- int * k = new int [3];
- k = new int [5]; //memory leak
- int * l = k;



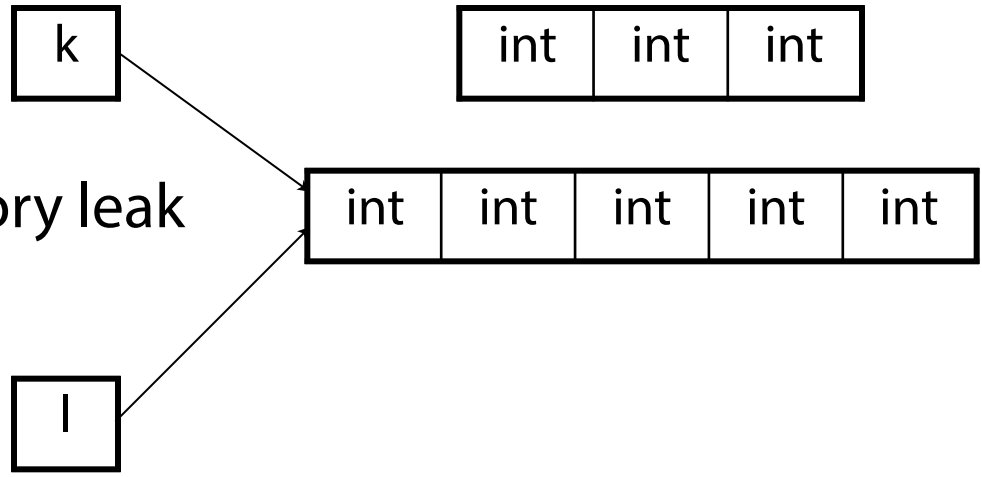
Dynamic Memory Allocation Dangers!

- int * k = new int [3];
- k = new int [5]; //memory leak
- int * l = k;



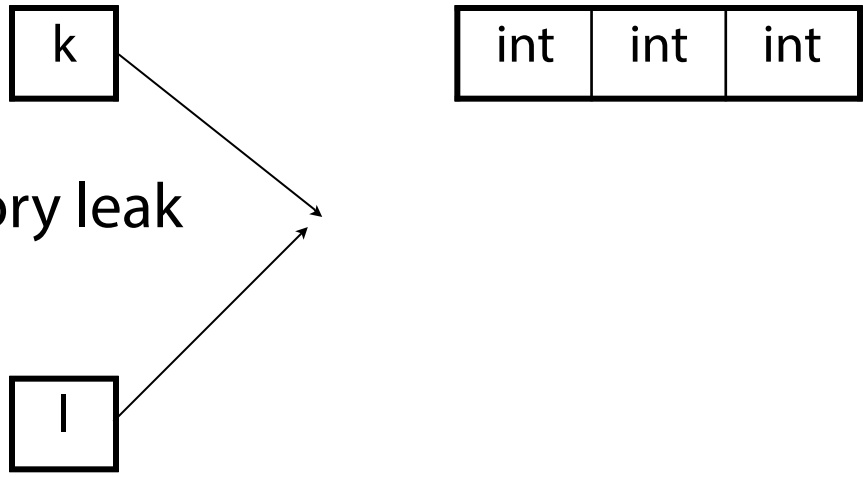
Dynamic Memory Allocation Dangers!

- int * k = new int [3];
- k = new int [5]; //memory leak
- int * l = k;
- delete [] k;



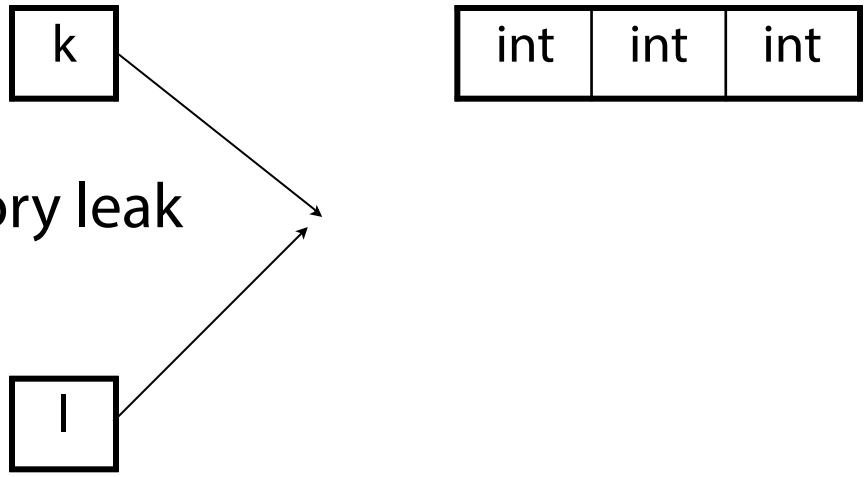
Dynamic Memory Allocation Dangers!

- 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

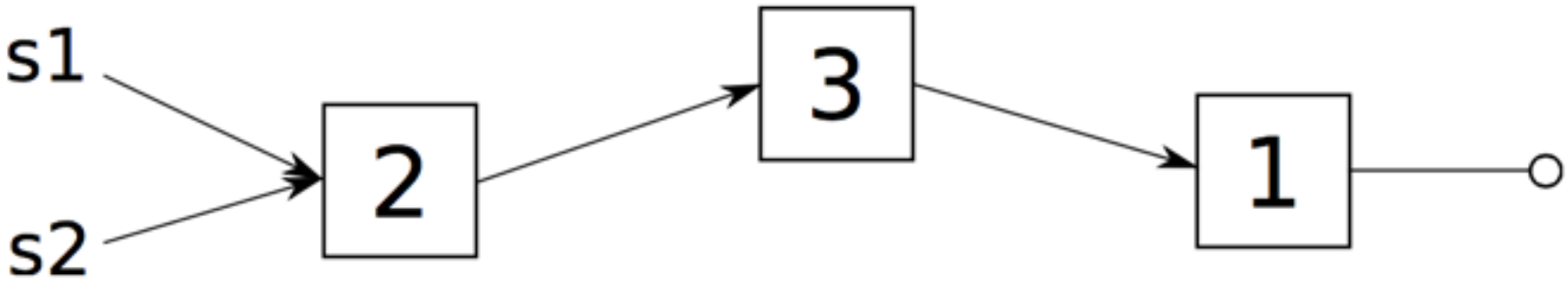
- Compiler creates a default copy constructor
- The default copy constructor will construct new a object and copy data members over.

```
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)
```

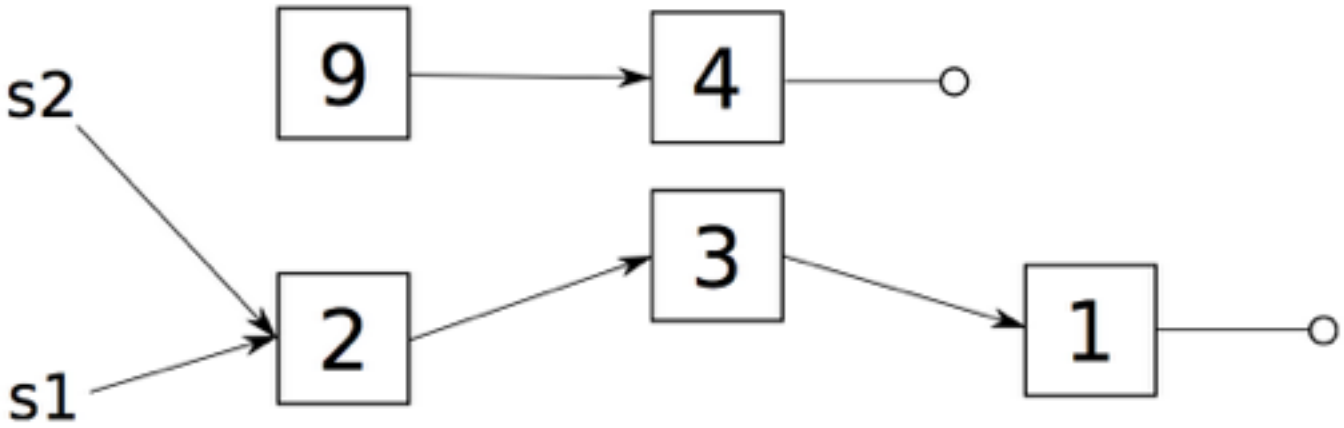
- Calling by value means using the copy constructor - infinite recursion.

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[]`.

(From: Script Exercise 158.a)

Dynamic Storage Solution

- Idea:

(From: Script Exercise 158.a)

Dynamic Storage Solution

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

(From: Script Exercise 158.a)

Dynamic Storage Solution

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

(From: Script Exercise 158.a)

Dynamic Storage Solution

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

(From: Script Exercise 158.a)

Dynamic Storage Solution

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

(From: Script Exercise 158.a)

Dynamic Storage Solution

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

(From: Script Exercise 158.a)

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

(From: Script Exercise 158.a)

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

(From: Script Exercise 158.a)

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

(From: Script Exercise 158.a)

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

(From: Script Exercise 158.a)

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[]`)

(From: Script Exercise 158.a)

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

(From: Script Exercise 158.a)

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

(From: Script Exercise 158.a)

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

(From: Script Exercise 158.a)

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

(From: Script Exercise 158.a)

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 rang
  5. Go back to 2. wit mory

(From: Script Exercise 158.a)

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
```


(From: Script Exercise 158.a)

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…
  - ```
    length_n = length_o * 2
    length_n = length_o + 2
    ```


(From: Script Exercise 158.a)
    ```

Dynamic Storage Solution

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

elements	Case a)	Case b)

(From: Script Exercise 158.a)

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

(From: Script Exercise 158.a)

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.

(From: Script Exercise 158.a)

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.

(From: Script Exercise 158.a)

Dynamic Storage Solution

- Larger by: a) factor 2 b) co

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

hosen

Each resizing means:

Copy WHOLE array

ficantly fewer resizings.

Factor 2 is an arbitrary, but good choice.

(From: Script Exercise 158.a)

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!

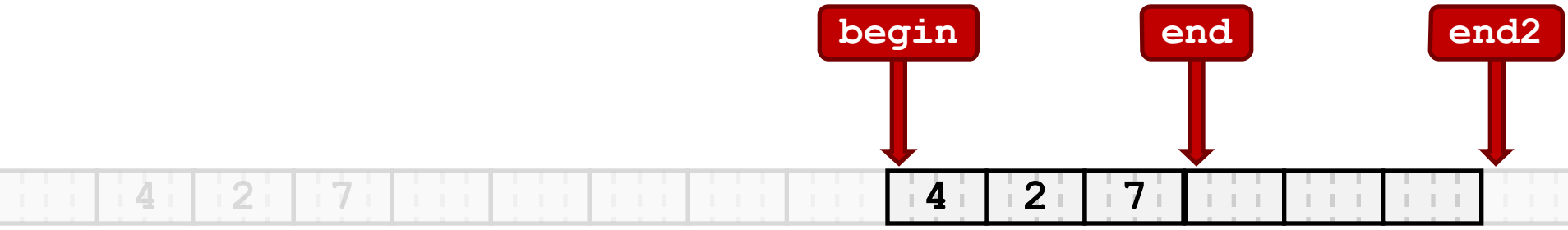
Dynamic Storage in Vectors

- Vectors store 3 pointers:

  `begin`: begin of memory

  `end`: end of *user-accessible* part

  `end2`: end of allocated part



Dynamic Storage in Vectors

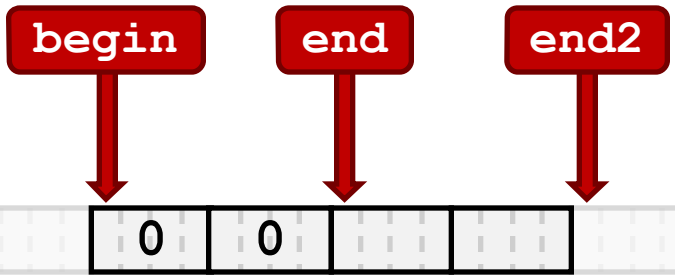
- Example:

```
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:

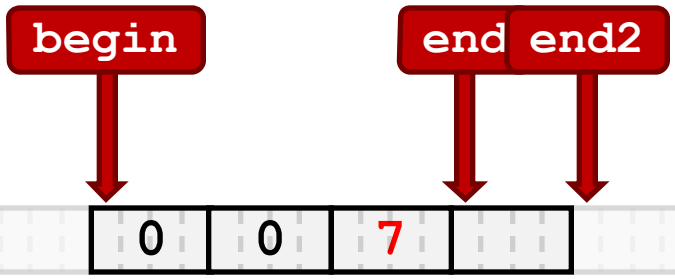
```
ifmp::vector<int> vec (2,0); // 0 0
vec.push_back(7);            // 0 0 7
vec.push_back(2);            // 0 0 7 2
vec.push_back(6);            // 0 0 7 2 6
```



Dynamic Storage in Vectors

- Example:

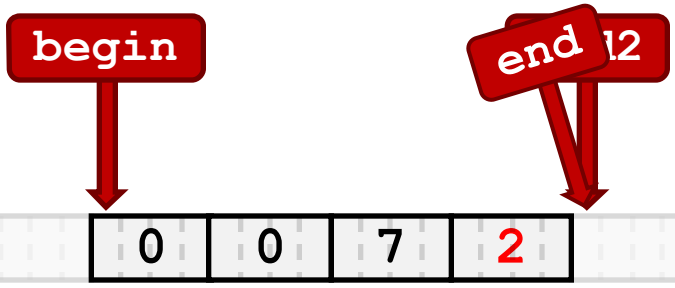
```
ifmp::vector<int> vec (2,0); // 0 0
vec.push_back(7);            // 0 0 7
vec.push_back(2);            // 0 0 7 2
vec.push_back(6);            // 0 0 7 2 6
```



Dynamic Storage in Vectors

- Example:

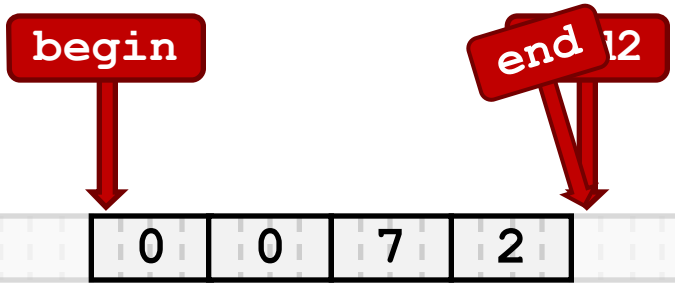
```
ifmp::vector<int> vec (2,0); // 0 0
vec.push_back(7);            // 0 0 7
vec.push_back(2);            // 0 0 7 2
vec.push_back(6);            // 0 0 7 2 6
```



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

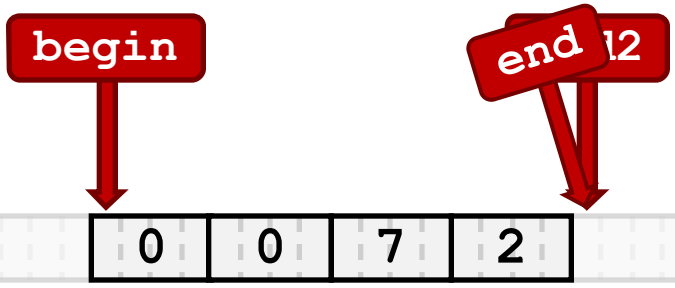
- Example:

```
ifmp::vector<
vec.push_ba
vec.push_b
vec.push_b
```

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

Space full

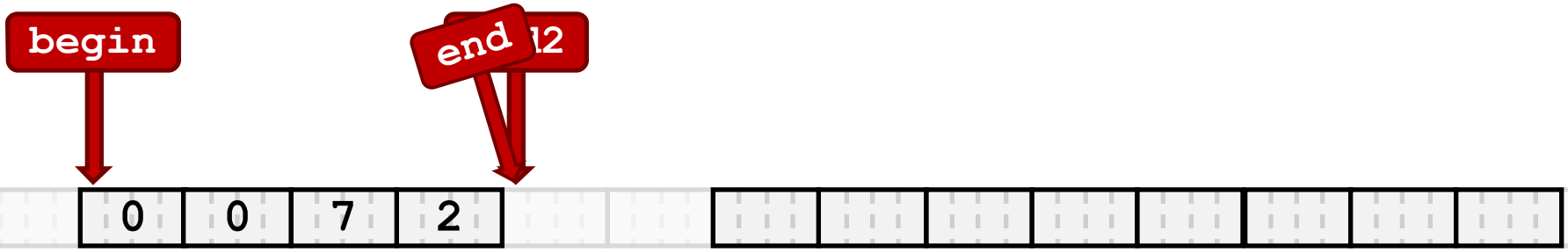
Now:
copy range



Dynamic Storage in Vectors

- Example:

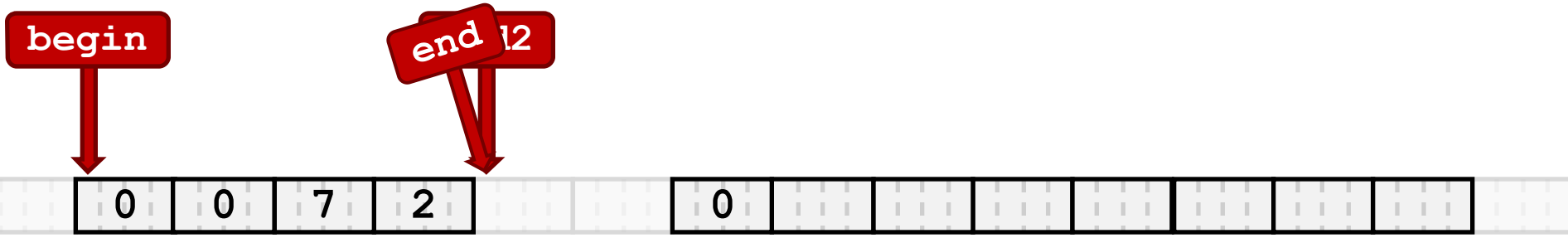
```
ifmp::vector<int> vec (2,0); // 0 0
vec.push_back(7);            // 0 0 7
vec.push_back(2);            // 0 0 7 2
vec.push_back(6);            // 0 0 7 2 6
```



Dynamic Storage in Vectors

- Example:

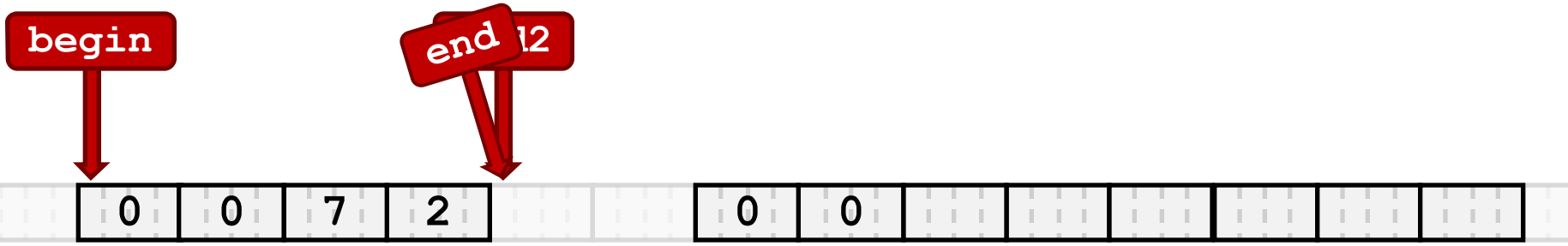
```
ifmp::vector<int> vec (2,0); // 0 0
vec.push_back(7);            // 0 0 7
vec.push_back(2);            // 0 0 7 2
vec.push_back(6);            // 0 0 7 2 6
```



Dynamic Storage in Vectors

- Example:

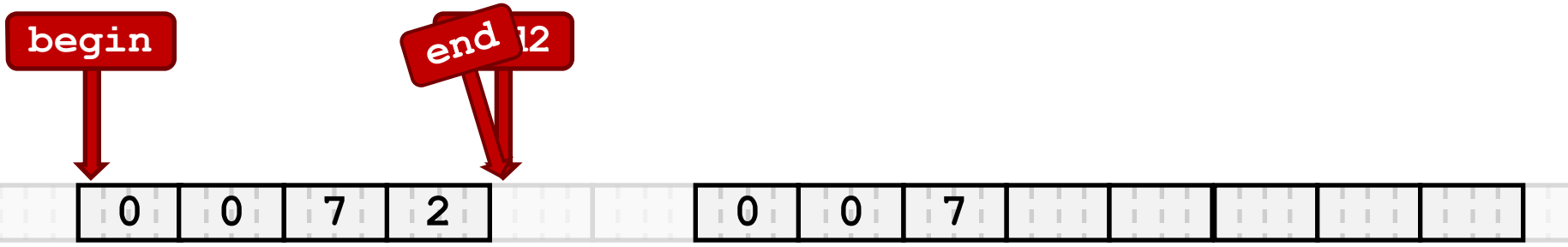
```
ifmp::vector<int> vec (2,0); // 0 0
vec.push_back(7);            // 0 0 7
vec.push_back(2);            // 0 0 7 2
vec.push_back(6);            // 0 0 7 2 6
```



Dynamic Storage in Vectors

- Example:

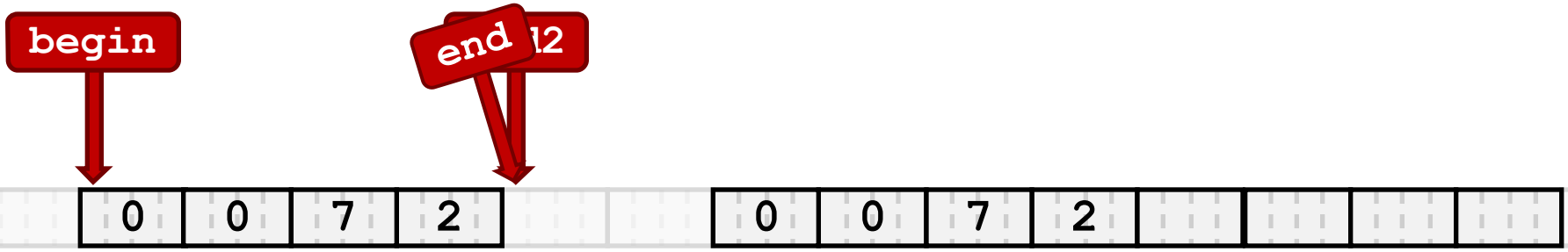
```
ifmp::vector<int> vec (2,0); // 0 0
vec.push_back(7);            // 0 0 7
vec.push_back(2);            // 0 0 7 2
vec.push_back(6);            // 0 0 7 2 6
```



Dynamic Storage in Vectors

- Example:

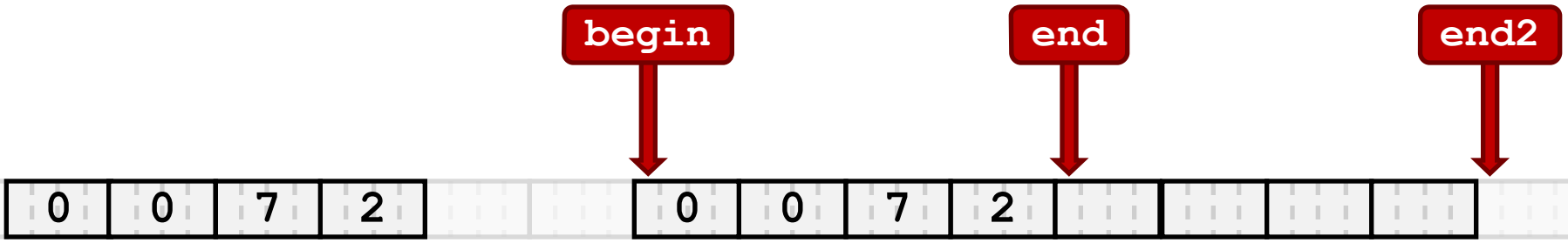
```
ifmp::vector<int> vec (2,0); // 0 0
vec.push_back(7);            // 0 0 7
vec.push_back(2);            // 0 0 7 2
vec.push_back(6);            // 0 0 7 2 6
```



Dynamic Storage in Vectors

- Example:

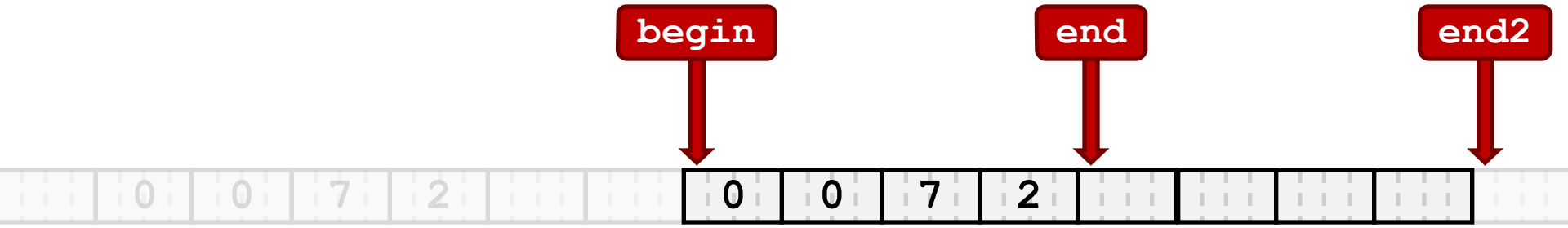
```
ifmp::vector<int> vec (2,0); // 0 0
vec.push_back(7);            // 0 0 7
vec.push_back(2);            // 0 0 7 2
vec.push_back(6);            // 0 0 7 2 6
```



Dynamic Storage in Vectors

- Example:

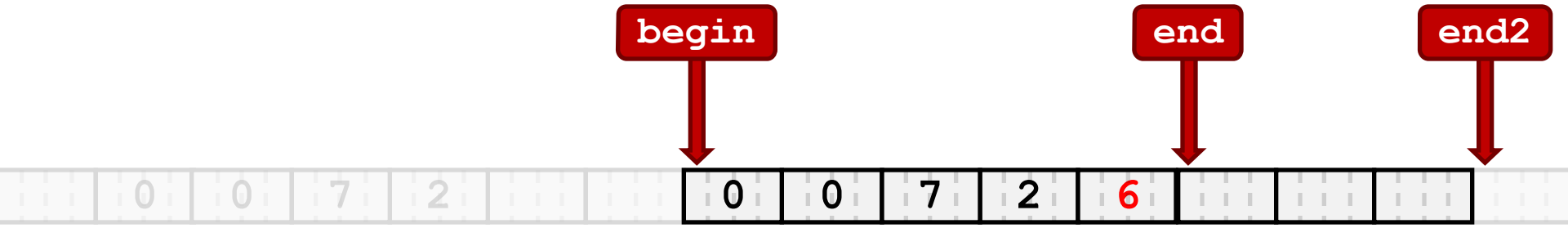
```
ifmp::vector<int> vec (2,0); // 0 0
vec.push_back(7);            // 0 0 7
vec.push_back(2);            // 0 0 7 2
vec.push_back(6);            // 0 0 7 2 6
```



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