# Wie gut fühlt Ihr Euch mit dem Stoff?

Siehe grobe Themenübersicht:
`https://n.ethz.ch/~pochs/info1/exam_overview.pdf`

- recursion
- references vs. pointers
- classes and structs
- function- and operator-overloading

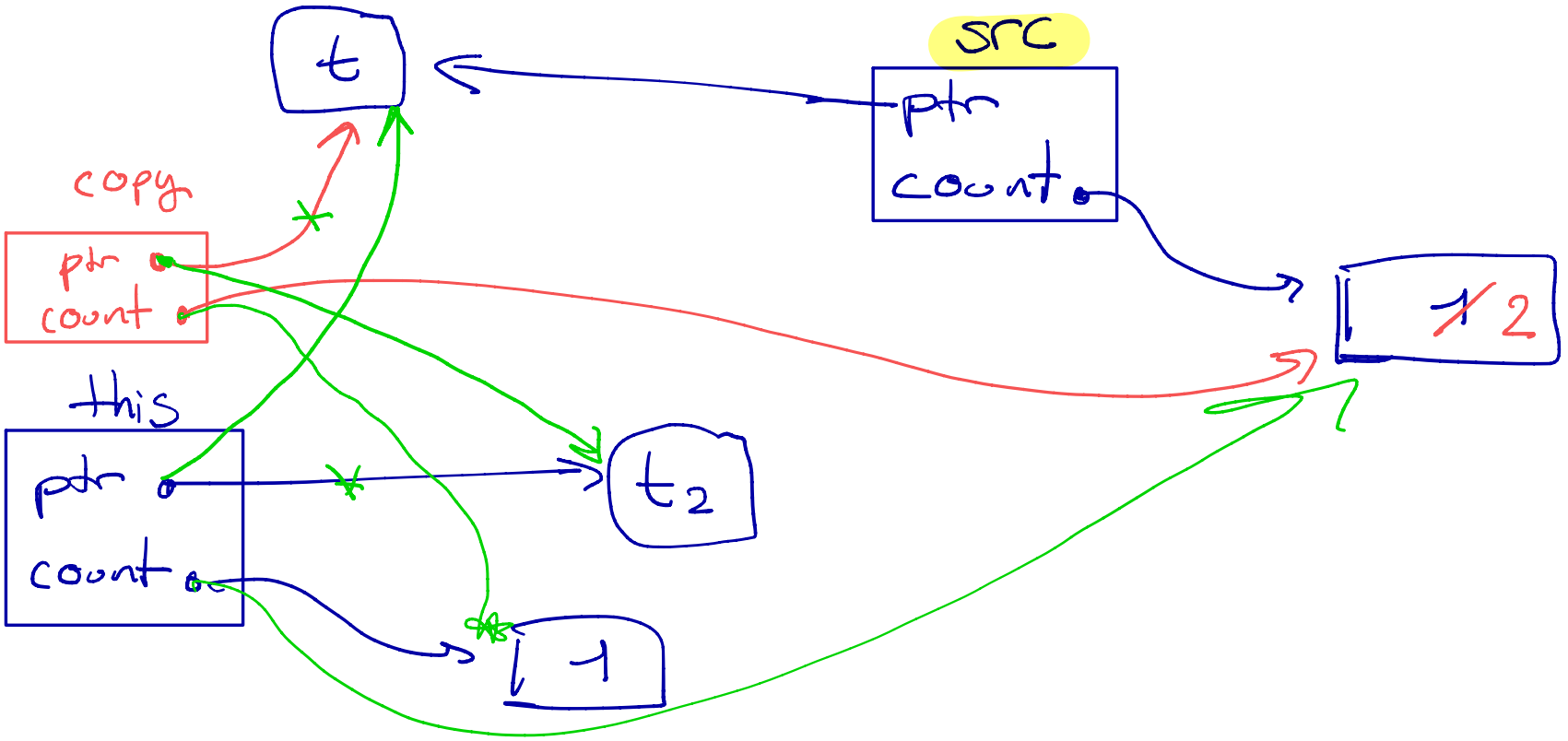
Allg. Hinweis im Hinblick auf die Prüfung: Versucht wirklich möglichst nah an der Aufgabe zu bleiben. E.g. wenn eine Funktion etwas returnen soll und dann in der main dieser return Wert geprinted wird: versucht wirklich auch das richtige zu returnen und nicht einfach selber irgendwas zu printen.

# Fragen zu der Serie?

# Trick beim Smart::operator=

```
27 Smart& Smart::operator=(const Smart& src) {
28   if (ptr != src.ptr) {
29     Smart copy(src);
30     // std::swap swaps the content of two variables, see also
31     //   https://en.cppreference.com/w/cpp/algorithm/swap
32     // This can of course also be done using a temporary local variable.
33     std::swap(count, copy.count);
34     std::swap(ptr, copy.ptr);
35   }
36   return *this;
37 }
```

S1 = S2
"this" "src"



# Dynamic Memory Allocation: new

```
// runde Klammern:
// - wir wollen Speicher für einen einzigen Integer
// - p zeigt auf diesen int
int *p = new int(int_value);

// eckige Klammern:
// - wir wollen Speicher für num_ints Integer
// - p zeigt auf den ersten int
int *p = new int[num_ints];
```

int[2]{0, 1}

# Dynamic Memory Allocation: delete

```
// Speicher für ein Integer allozieren:
int *p = new int(int_value);
// Speicher für ein Integer löschen:
delete p;

// Speicher für einen Block allozieren:
int *p = new int[num_ints];
// Speicher für einen Block löschen:
delete[] p;
```

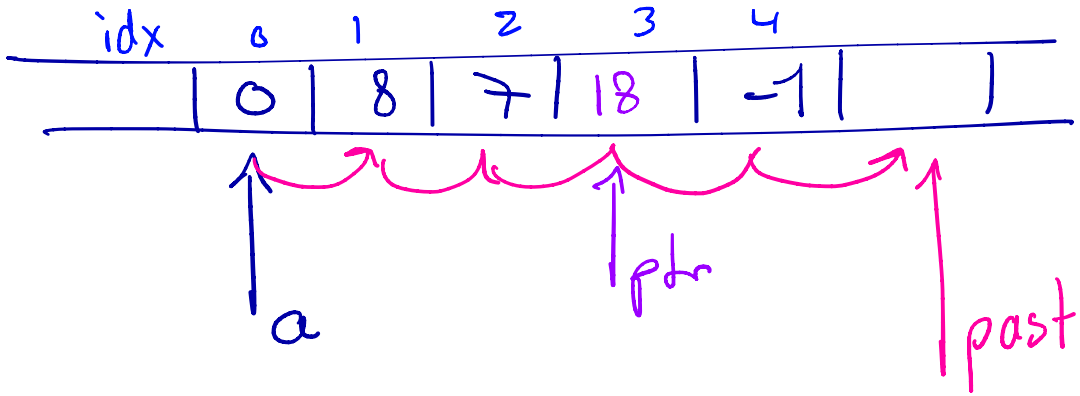
# Dynamic Memory Allocation: delete

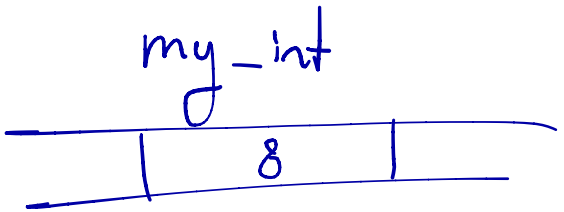
## Grobe Regel:

- Wenn wir Variablen/Objekte mit `new` deklarieren, dann haben wir die ganze Kontrolle darüber, wann das Objekt kreiert und zerstört wird. Insbesondere gehen sie nicht "out-of-scope" so wie wir es sonst gesehen haben. (Heisst aber auch wir müssen uns darum kümmern.)
- Der ganze Sinn von dynamic memory ist ja, die volle Kontrolle über die Lebenszeit der Objekte zu haben.
- Für jedes `new` ein `delete` und für jedes `new[]` ein `delete[]`.

# Pointer Program

```
int* a = new int[5]{0, 8, 7, 2, -1};
int* ptr = a;                                  // pointer assignment
++ptr;                                         // shift to the right
int my_int = *ptr;                             // read target
ptr += 2;                                      // shift by 2 elements
*ptr = 18;                                     // overwrite target
int* past = a+5;
std::cout << (ptr < past) << "\n"; // compare pointers
```

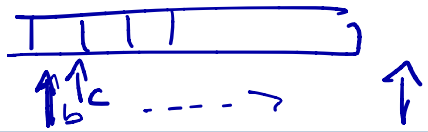



a[2]  *(a+2)

# Pointer Program

Find and fix at least 3 problems in the following program.




```
#include <iostream>
int main () {
    int* a = new int[7]{0, 6, 5, 3, 2, 4, 1};
    int* b = new int[7];
    int* c = b;
    // copy a into b using pointers
    for (int* p = a; p <= a+7; ++p) {
        *c++ = *p;
    }
    // cross-check with random access
    for (int i = 0; i <= 7; ++i) {
        if (a[i] != c[i]) {
            std::cout << "Oops, copy error...\n";
        }
    }
    return 0;
}
```

*c++ ≙ *(c++)
d.h. zuerst c++ und dann deref.
ABER: c++ ist "use then change" d.h. wir machen tatsächlich *c = *p und haben als "Nebeneffekt" das c+=1 gerechnet wird.

# Pointer Program

```
#include <iostream>
int main () {
    int* a = new int[7]{0, 6, 5, 3, 2,
    int* b = new int[7];
    int* c = b;
    // copy a into b using pointers
    for (int* p = a; p <= a+7; ++p) {
        *c++ = *p;
    }
    // cross-check with random access
    for (int i = 0; i <= 7; ++i) {
        if (a[i] != c[i]) {
            std::cout << "Oops, copy error...\n";
        }
    }
    return 0;
}
```

p = a+7 is dereferenced

Solution:
Use < instead of <=

# Pointer Program

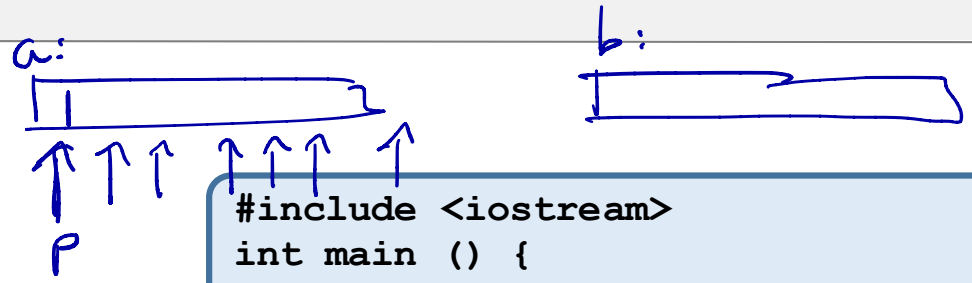
```
#include <iostream>
int main () {
    int* a = new int[7]{0, 6, 5, 3, 2, 4
    int* b = new int[7];
    int* c = b;
    // copy a into b using pointers
    for (int* p = a; p <= a+7; ++p) {
        *c++ = *p;
    }
    // cross-check with random access
    for (int i = 0; i <= 7; ++i) {
        if (a[i] != c[i]) {
            std::cout << "Oops, copy error
        }
    }
    return 0;
}
```

p = a+7 is dereferenced

Solution:
Use < instead of <=

Same problem as above

# Pointer Program




```
#include <iostream>
int main () {
    int* a = new int[7]{0, 6, 5, 3, 2, 4, 1};
    int* b = new int[7];
    int* c = b;
    // copy a into b using pointers
    for (int* p = a; p <= a+7; ++p) {
        *c++ = *p;

    // cross-check with random access
    for (int i = 0; i <= 7; ++i) {
        if (a[i] != c[i]) {
            std::cout << "Oops, copy err
        }
    }
    return 0;
}
```

`c` doesn't point to `b[0]` anymore.

**Solution:**
Use `b` instead of `c`

`p = a+7` is dereferenced

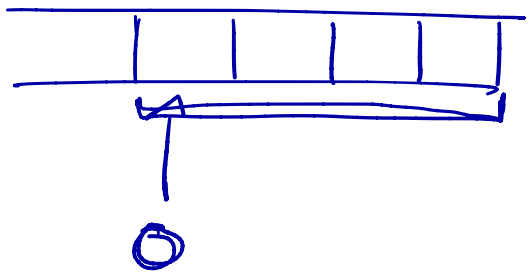
**Solution:**
Use `<` instead of `<=`

Same problem as above

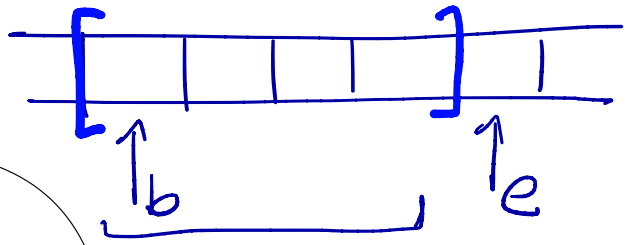
# Exercise – Applying Pointers

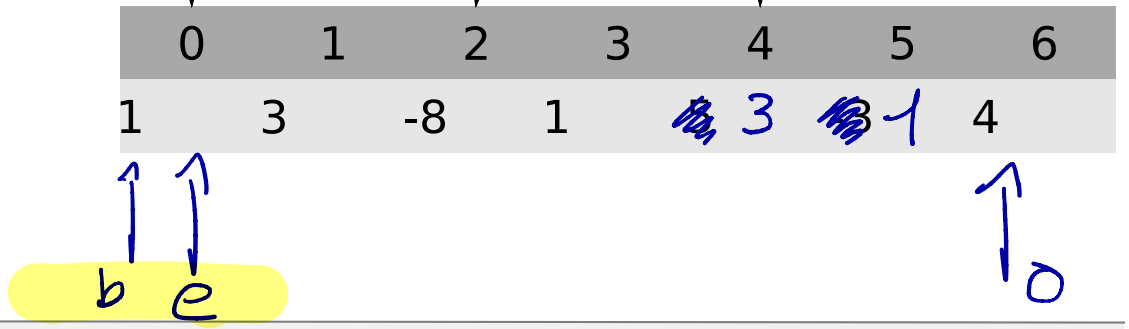
# Exercise – Applying Pointers

```
// PRE: [b, e) and [o, o+(e-b)) are disjoint
//      valid ranges
void f (int* b, int* e, int* o) {
    while (b != e) {
        --e;
        *o = *e;
        ++o;
    }
}
```




| Variable | Value |
|---|---|
| b | ● |
| e | ● |
| o | ● |



# Exercise – Applying Pointers

Now determine a POST-condition for the function.

```
// PRE: [b, e) and [o, o+(e-b)) are disjoint
//      valid ranges
void f (int* b, int* e, int* o) {
    while (b != e) {
        --e;
        *o = *e;
        ++o;
    }
}
```

(From: Script Exercise 113)

# Exercise – Applying Pointers

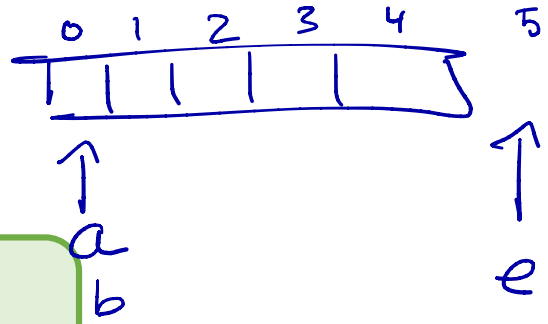
```
// PRE: [b, e) and [o, o+(e-b)) are disjoint
//      valid ranges
// POST: The range [b, e) is copied in reverse
//       order into the range [o, o+(e-b))
void f (int* b, int* e, int* o) {
    while (b != e) {
        --e;
        *o = *e;
        ++o;
    }
}
```

(From: Script Exercise 113)

# Exercise – Valid Inputs

# Exercise – Valid Inputs

- Which of these inputs are valid?




```
int* a = new int[5];
// Initialise a.
   a)   f(a, a+5, a+5);
   b)   f(a, a+2, a+3);
   c)   f(a, a+3, a+2);
```

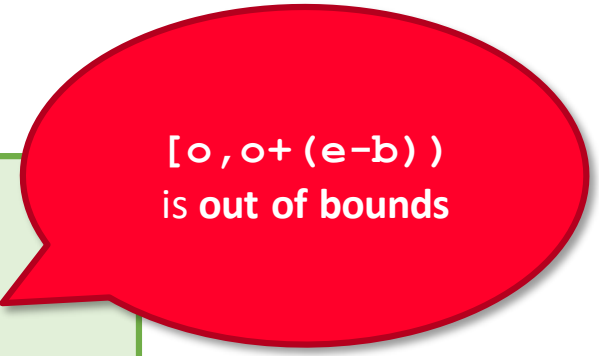
```
// PRE: [b, e) and [o, o+(e-b)) are disjoint
//      valid ranges
void f (int* b, int* e, int* o) {
    while (b != e) {
        --e;
        *o = *e;
        ++o;
    }
}
```

(From: Script Exercise 113)

# Exercise – Valid Inputs

- Which of these inputs are valid?

```
int* a = new int[5];
// Initialise a.
  a)  f(a, a+5, a+5);
  b)  f(a, a+2, a+3);
  c)  f(a, a+3, a+2);
```




```
// PRE: [b, e) and [o, o+(e-b)) are disjoint
//      valid ranges
void f (int* b, int* e, int* o) {
    while (b != e) {
        --e;
        *o = *e;
        ++o;
    }
}
```

(From: Script Exercise 113)

# Exercise – Valid Inputs

- Which of these inputs are valid?

```
int* a = new int[5];
// Initialise a.
  a)  f(a, a+5, a+5);
  b)  f(a, a+2, a+3);
  c)  f(a, a+3, a+2);
```

[o,o+(e-b)) is **out of bounds**

```
// PRE: [b, e) and [o, o+(e-b)) are disjoint
//      valid ranges
void f (int* b, int* e, int* o) {
    while (b != e) {
        --e;
        *o = *e;
        ++o;
    }
}
```

(From: Script Exercise 113)

# Exercise – Valid Inputs

- Which of these inputs are valid?

```
int* a = new int[5];
// Initialise a.
  a)  f(a, a+5, a+5);
  b)  f(a, a+2, a+3);
  c)  f(a, a+3, a+2);
```

[o,o+(e-b)) is **out of bounds**

Ranges **not disjoint**

```
// PRE: [b, e) and [o, o+(e-b)) are disjoint
//      valid ranges
void f (int* b, int* e, int* o) {
    while (b != e) {
        --e;
        *o = *e;
        ++o;
    }
}
```

(From: Script Exercise 113)

# Exercise – `const` Correctness

# Exercise – `const` Correctness

- Make the function `const`-correct.

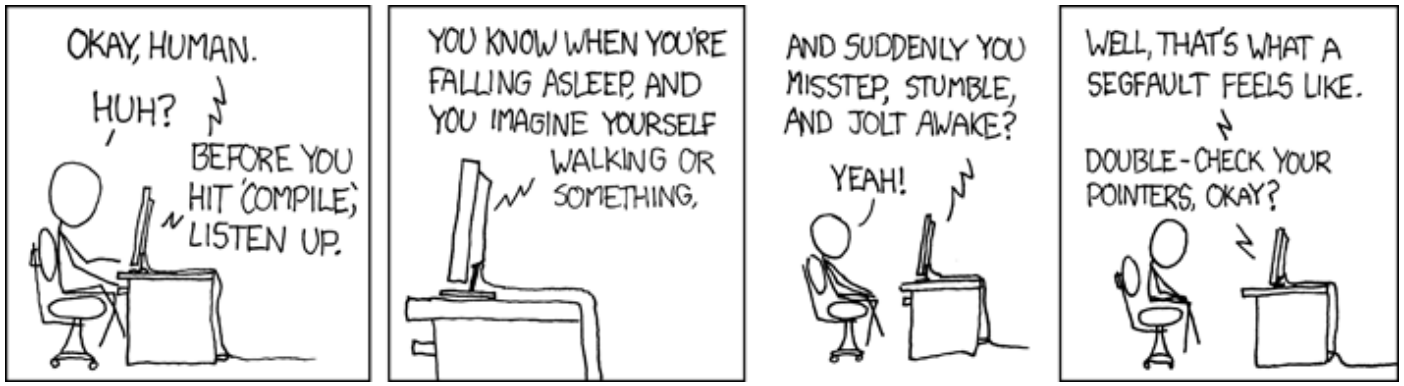
```
// PRE: [b, e) and [o, o+(e-b)) are disjoint
//      valid ranges
void f (int* b, int* e, int* o) {
    while (b != e) {
        --e;
        *o = *e;
        ++o;
    }
}
```

# Exercise – `const` Correctness

- Make the function `const`-correct.

```
// PRE: [b, e) and [o, o+(e-b)) are disjoint
//      valid ranges
void f (const int* const b, const int* e, int* o) {
    while (b != e) {
        --e;
        *o = *e;
        ++o;
    }
}
```

Informatik
Exercise Session 13




https://xkcd.com/371/

Find mistakes in the following code and suggest fixes:

```
1  // PRE: len is the length of the memory block that starts at array
2  void test1(int* array, int len) {
3      int* fourth = array + 3;
4      if (len > 3) {
5          std::cout << *fourth << std::endl;
6      }
7      for (int* p = array; p != array + len; ++p) {
8          std::cout << *p << std::endl;
9      }
10 }
```

Find mistakes in the following code and suggest fixes:

```
1  // PRE: len is the length of the memory block that starts at array
2  void test1(int* array, int len) {
3      //int* fourth = array + 3;       // ERROR
4      if (len > 3) {
5          int* fourth = array + 3;     // OK
6          std::cout << *fourth << std::endl;
7      }
8      for (int* p = array; p != array + len; ++p) {
9          std::cout << *p << std::endl;
10     }
11 }
```

Even if the pointer is not dereferenced, it must point into a memory block or to the element just after its end.

Find mistakes in the following code and suggest fixes:

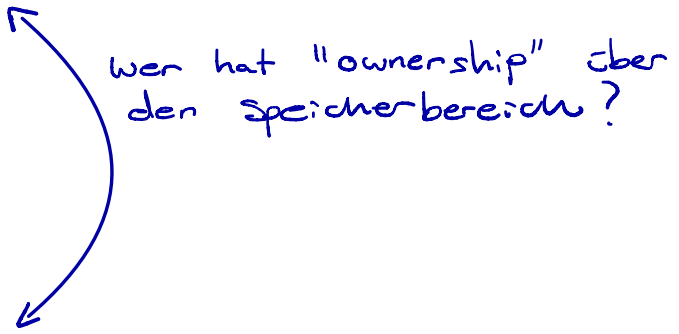
```
1  // PRE: len >= 2
2  int* fib(unsigned int len) {
3      int* array = new int[len];
4      array[0] = 0; array[1] = 1;
5      for (int* p = array+2; p < array + len; ++p) {
6          *p = *(p-2) + *(p-1); }
7      return array; }
8  void print(int* array, int len) {
9      for (int* p = array+2; p < array + len; ++p) {
10         std::cout << *p << " ";
11     }
12 }
13 void test2(unsigned int len) {
14     int* array = fib(len);
15     print(array, len);
16 }
```

```
1  // PRE: len >= 2
2  int* fib(unsigned int len) {
3      int* array = new int[len];
4      array[0] = 0; array[1] = 1;
5      for (int* p = array+2; p < array + len; ++p) {
6          *p = *(p-2) + *(p-1); }
7      return array; }
8  void print(int* array, int len) {
9      for (int* p = array+2; p < array + len; ++p) {
10         std::cout << *p << " ";
11     }
12 }
13 void test2(unsigned int len) {
14     int* array = fib(len);
15     print(array, len);
16 }   // array is leaked; to fix add: delete[] array
```

Find mistakes in the following code and suggest fixes:

```
1  // PRE: len >= 2
2  int* fib(unsigned int len) {
3      // ...
4  }
5  void print(int* m, int len) {
6      for (int* p = m+2; p < m + len; ++p) {
7          std::cout << *p << " ";
8      }
9      delete m;
10 }
11 void test2(unsigned int len) {
12     int* array = fib(len);
13     print(array, len);
14     delete[] array;
15 }
```

```
1  // PRE: len >= 2
2  int* fib(unsigned int len) {
3      // ...
4  }
5  void print(int* m, int len) {
6      for (int* p = m+2; p < m + len; ++p) {
7          std::cout << *p << " ";
8      }
9      delete m;   // should be delete[]
10 }
11 void test2(unsigned int len) {
12     int* array = fib(len);
13     print(array, len);
14     delete[] array; // array deallocated twice
15 }
```



# Push Back

std::vector< >

.size()

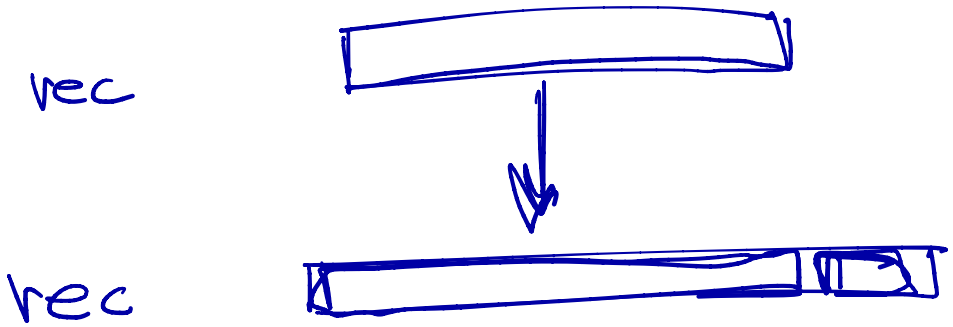
```
class our_vector {
  unsigned int count;    "size"
  int* elements;

public:
  our_vector();
  // POST: this contains the same sequence as before with the
  // new_element appended at the end.
  void push_back(int new_element);

  // POST: prints the content of the sequence
  void print(std::ostream& sink) const;
}
```

# Push Back: Idee

new int[len]



# Push Back

```
void copy_range(const int* const source_begin, const int* const source_end,
                int* const destination_begin) {
  int* dst = destination_begin;
  for (const int* src = source_begin; src != source_end; ++src) {
    *dst = *src;
    ++dst;
  }
}
```

.at(i) ≙ [i]

```
// POST: this contains the same sequence as before with the
// new_element appended at the end.
void our_vector::push_back(int new_element) {
  int* const new_elements = new int[this->count + 1];
  copy_range(this->elements, this->elements + this->count, new_elements);
  delete[] this->elements;
  new_elements[this->count] = new_element; // *(new_elements + (*this).count)
  this->count++;
  this->elements = new_elements;
}
```