ETH zürich

Übungsstunde W13

Informatik (RW & CBB & Statistik) – HS 23

Übersicht

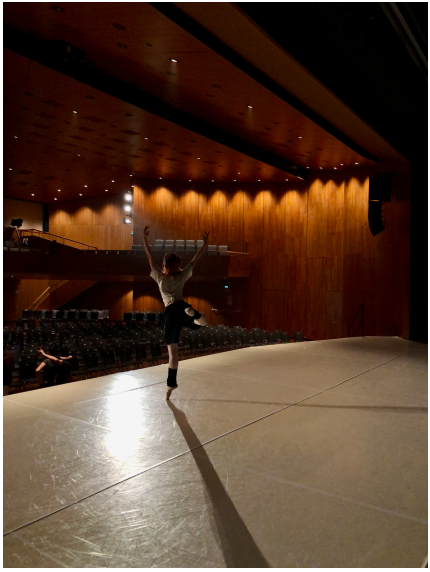
Heutiges Programm

Where's the TA?
Follow-up
Ziele
Pointers
Aufgabe "Push Back"
Memory Management
Outro

n.ethz.ch/~agavranovic

Wo ist Lily?

- Lily hat Intensivprobenwoche für den Nussknacker
- und ist erst nächste Woche wieder da
- und schickt ganz liebe Grüsse



2. Follow-up

Follow-up aus vorherigen Übungsstunden

Follow-up aus vorherigen Übungsstunden

- Wie schon angekündigt machen wir nächste Woche ein Semester-Recap mit Themen, die euch noch unklar sind

Follow-up aus vorherigen Übungsstunden

- Wie schon angekündigt machen wir nächste Woche ein Semester-Recap mit Themen, die euch noch unklar sind
- Schreibt sehr gerne konkrete Fragen ins Google Form auf Lily's homepage

3. Ziele

Ziele

- ☐ Den Unterschied verstehen zwischen `new` / `delete` und `new[]` / `delete[]`
- ☐ Programme tracen können, die Pointer Arithmetik benutzen
- ☐ Programme schreiben können, die Pointer Arithmetik benutzen

4. Pointers

`new` VS `new[]`

- `new T` alloziert **eine** Speicherstelle für den Typ `T`

[1] diese Speicherstellen werden *zusammenhängend* sein, d. h. "nebeneinander" im Speicher

`new` VS `new[]`

- `new T` alloziert **eine** Speicherstelle für den Typ `T`
- `new T[n]` alloziert **n** Speicherstellen für den Typ `T`[1]

[1] diese Speicherstellen werden *zusammenhängend* sein, d. h. "nebeneinander" im Speicher

new VS new[]

- `new T` alloziert **eine** Speicherstelle für den Typ `T`
- `new T[n]` alloziert **n** Speicherstellen für den Typ `T`[1]
- Beide geben einen Pointer zurück, bei einer Range zeigt dieser auf das erste Objekt

ptr

[1] diese Speicherstellen werden *zusammenhängend* sein, d. h. "nebeneinander" im Speicher

Arrays

Statisch allozierter Array

Arrays

Statisch allozierter Array

```
int myStatArr[3] = {2, 3, 8};
```

- `myStatArr` zeigt nun auf die

Arrays

Statisch allozierter Array

```
int myStatArr[3] = {2, 3, 8};
```

- `myStatArr` zeigt nun auf die `2`
- `*myStatArr` gibt

Arrays

Statisch allozierter Array

```
int myStatArr[3] = {2, 3, 8};
```

- `myStatArr` zeigt nun auf die `2`
- `*myStatArr` gibt `2`
- `myStatArr[2]` gibt

Arrays

Statisch allozierter Array

```
int myStatArr[3] = {2, 3, 8};
```

- `myStatArr` zeigt nun auf die 2
- `*myStatArr` gibt 2
- `myStatArr[2]` gibt 8
- `myStatArr[1] = -4;`

Arrays

Statisch allozierter Array

```
int myStatArr[3] = {2, 3, 8};
```

- `myStatArr` zeigt nun auf die `2`
- `*myStatArr` gibt `2`
- `myStatArr[2]` gibt `8`
- `myStatArr[1] = -4` setzt `3` auf `-4`

Arrays

Statisch allozierter Array

```
int myStatArr[3] = {2, 3, 8};
```

- `myStatArr` zeigt nun auf die `2`
- `*myStatArr` gibt `2`
- `myStatArr[2]` gibt `8`
- `myStatArr[1] = -4` setzt `3` auf `-4`

Dynamisch allozierterer Array

Arrays

Statisch allozierter Array

```
int myStatArr[3] = {2, 3, 8};
```

- `myStatArr` zeigt nun auf die `2`
- `*myStatArr` gibt `2`
- `myStatArr[2]` gibt `8`
- `myStatArr[1] = -4` setzt `3` auf `-4`

Dynamisch allozierterer Array

```
int* myDynArr = new int[3]{2, 3, 8};
```

Arrays

Statisch allozierter Array

```
int myStatArr[3] = {2, 3, 8};
```

- `myStatArr` zeigt nun auf die `2`
- `*myStatArr` gibt `2`
- `myStatArr[2]` gibt `8`
- `myStatArr[1] = -4` setzt `3` auf `-4`

Dynamisch allozierterer Array

```
int* myDynArr = new int[3]{2, 3, 8};
```

- `myDynArr` zeigt nun auf die

Arrays

Statisch allozierter Array

```
int myStatArr[3] = {2, 3, 8};
```

- `myStatArr` zeigt nun auf die 2
- `*myStatArr` gibt 2
- `myStatArr[2]` gibt 8
- `myStatArr[1] = -4` setzt 3 auf `-4`

Dynamisch allozierterer Array

```
int* myDynArr = new int[3]{2, 3, 8};
```

- `myDynArr` zeigt nun auf die 2
- `*myDynArr` gibt

Arrays

Statisch allozierter Array

```
int myStatArr[3] = {2, 3, 8};
```

- `myStatArr` zeigt nun auf die 2
- `*myStatArr` gibt 2
- `myStatArr[2]` gibt 8
- `myStatArr[1] = -4` setzt 3 auf -4

Dynamisch allozierterer Array

```
int* myDynArr = new int[3]{2, 3, 8};
```

- `myDynArr` zeigt nun auf die 2
- `*myDynArr` gibt 2
- `myDynArr[2]` gibt

↑
brackets
dereferenzieren
für uns :)

Arrays

Statisch allozierter Array

```
int myStatArr[3] = {2, 3, 8};
```

- `myStatArr` zeigt nun auf die `2`
- `*myStatArr` gibt `2`
- `myStatArr[2]` gibt `8`
- `myStatArr[1] = -4` setzt `3` auf `-4`

Dynamisch allozierterer Array

```
int* myDynArr = new int[3]{2, 3, 8};
```

- `myDynArr` zeigt nun auf die `2`
- `*myDynArr` gibt `2`
- `myDynArr[2]` gibt `8`
- `myDynArr[1] = -4`

Arrays

Statisch allozierter Array

```
int myStatArr[3] = {2, 3, 8};
```

- `myStatArr` zeigt nun auf die 2
- `*myStatArr` gibt 2
- `myStatArr[2]` gibt 8
- `myStatArr[1] = -4` setzt 3 auf -4

Dynamisch allozierterer Array

```
int* myDynArr = new int[3]{2, 3, 8};
```

- `myDynArr` zeigt nun auf die 2
- `*myDynArr` gibt 2
- `myDynArr[2]` gibt 8
- `myDynArr[1] = -4` setzt 3 auf -4

`delete` vs `delete[]`

- Es gilt weiterhin: zu jedem **`new`** ein **`delete`**

`delete` vs `delete[]`

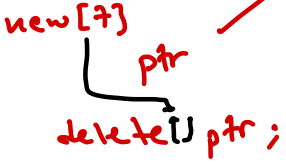
- Es gilt weiterhin: zu jedem **`new`** ein **`delete`**
- **`delete[]`** ist der entsprechende Operator zu **`new[]`**

`delete` vs `delete[]`

- Es gilt weiterhin: zu jedem **`new`** ein **`delete`**
- **`delete[]`** ist der entsprechende Operator zu **`new[]`**
- Auch hier aufpassen: Wir löschen nicht den Pointer, sondern die Range an Objekten, auf die der Pointer zeigt

delete vs delete[]

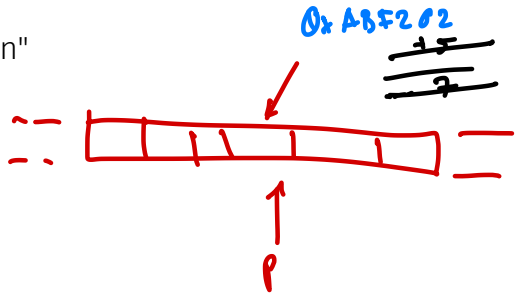
- Es gilt weiterhin: zu jedem `new` ein `delete`
- `delete[]` ist der entsprechende Operator zu `new[]`
- Auch hier aufpassen: Wir löschen nicht den Pointer, sondern die Range an Objekten, auf die der Pointer zeigt
- **Häufige Fehlerquelle**
  der Aufruf von `delete` für das erste Element, aber nicht für das gesamte Array (mit `delete[]`)



Pointer Arithmetik

- Wir können mit Pointern "rechnen"
- Die wichtigsten Befehle sind:

```
T* newp = p + 5;
```



Pointer Arithmetik

- Wir können mit Pointern "rechnen"
- Die wichtigsten Befehle sind:

- Temporäre Shifts

```
ptr + 3
ptr - 3
```

Pointer Arithmetik

- Wir können mit Pointern "rechnen"
- Die wichtigsten Befehle sind:

- Temporäre Shifts

```
ptr + 3
ptr - 3
```

- Permanente Shifts

```
ptr++
--ptr
ptr += 2
```

Pointer Arithmetik

n = e - a & off by one errors!

- Wir können mit Pointern "rechnen"
- Die wichtigsten Befehle sind:

- Temporäre Shifts

```
ptr + 3
ptr - 3
```

- Permanente Shifts

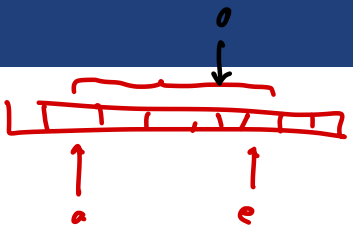
```
ptr++
--ptr;
ptr += 2
```

- Distanz zwischen Pointern bestimmen

```
ptr_1 - ptr_2
```

*(--p) ≠ *(p--)

Pointer Arithmetik



- Wir können mit Pointern "rechnen"
- Die wichtigsten Befehle sind:

- Temporäre Shifts

```
ptr + 3
ptr - 3
```

- Permanente Shifts

```
ptr++
--ptr
ptr += 2
```

- Distanz zwischen Pointern bestimmen

```
(ptr_1 - ptr_2)
```

returns an int !

- Positionen vergleichen

```
ptr_1 < ptr_2
ptr_1 != ptr_2
```

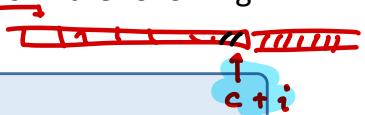
Fragen/Unklarheiten?

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
```

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;
}
```

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 `<=`

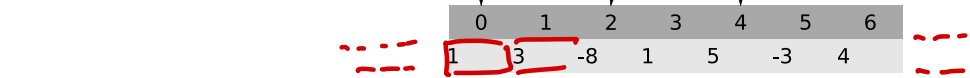
Same problem as above

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	●



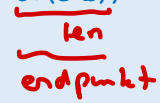
Exercise – Applying Pointers

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;
    }
}
```

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;
    }
}
```

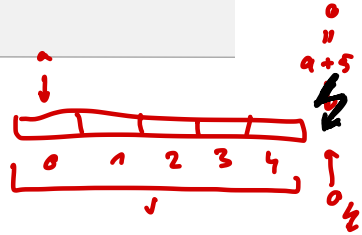


Exercise – Valid Inputs

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);
```



nicht 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;
    }
}
```

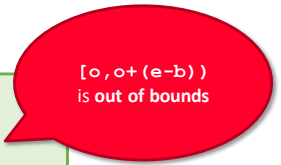
[a : incl. a
(a : ohne a

e - b = len

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;
    }
}
```

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;
    }
}
```

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;
    }
}
```

Ranges **not disjoint**

Fragen/Unklarheiten?

Pointer Costness

Es gibt zwei Arten von Constness bei Pointern:

```
const int* ptr = &a;
```

Pointer Costness

Es gibt zwei Arten von Constness bei Pointern:

```
const int* ptr = &a;
```

kein Schreibzugriff auf Target **a**

Pointer Costness

Es gibt zwei Arten von Constness bei Pointern:

```
const int* ptr = &a;
```

kein Schreibzugriff auf Target **a**

d.h. wir dürfen den Wert des Integers **a** *nicht* verändern

Pointer Costness

Es gibt zwei Arten von Constness bei Pointern:

```
const int* ptr = &a;
```

kein Schreibzugriff auf Target **a**

d.h. wir dürfen den Wert des Integers **a** *nicht* verändern

```
int* const ptr = &a;
```

Pointer Costness

Es gibt zwei Arten von Constness bei Pointern:

```
const int* ptr = &a;
```

kein Schreibzugriff auf Target **a**
d.h. wir dürfen den Wert des Integers **a** *nicht* verändern

```
int* const ptr = &a;
```

kein Schreibzugriff auf Pointer **ptr**

Pointer Costness

Es gibt zwei Arten von Constness bei Pointern:

```
const int* ptr = &a;
```

kein Schreibzugriff auf Target **a**

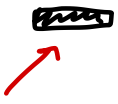
d.h. wir dürfen den Wert des Integers **a** *nicht* verändern

```
int* const ptr = &a;
```

kein Schreibzugriff auf Pointer **ptr**

d.h. wir dürfern nicht ändern, wohin der Pointer zeigt

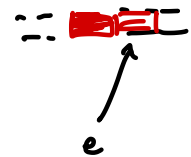
const



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;
    }
}
```

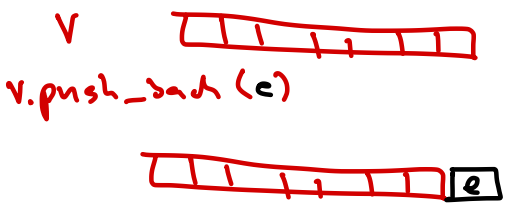
Exercise – `const` Correctness

- 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;
    }
}
```

Fragen/Unklarheiten?

5. Aufgabe "Push Back"



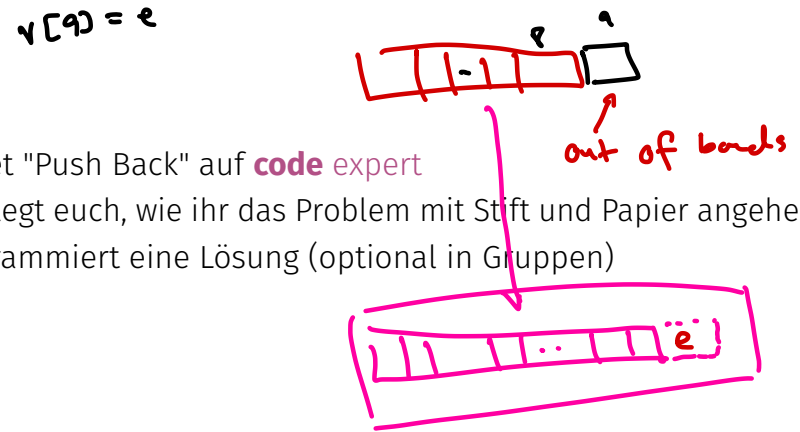
Aufgabe "Push Back"

- Öffnet "Push Back" auf **code** expert

Aufgabe "Push Back"

- Öffnet "Push Back" auf **code** expert
- Überlegt euch, wie ihr das Problem mit Stift und Papier angehen würdet

Aufgabe "Push Back"



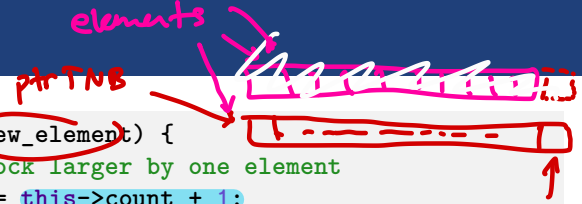
- Öffnet "Push Back" auf **code** expert
- Überlegt euch, wie ihr das Problem mit Stift und Papier angehen würdet
- Programmiert eine Lösung (optional in Gruppen)

Lösung zu "Push Back"

```
// PRE: source_begin points to first element to be copied.
// PRE: source_ends points to element after the last element to be copied.
// PRE: destination_begin points to first element of target memory block
// PRE: #elements in target memory location >= #elements in source
// POST: copies the content of the source memory block to the destination
//       memory block.
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;
  }
}
```

Lösung zu "Push Back"




```
void our_vector::push_back(int new_element) {
  // 1. Allocate a new memory block larger by one element
  unsigned int lenghtOfNewBlock = this->count + 1;
  int* const ptrToNewBlock = new int[lenghtOfNewBlock];

  // 2. Copy all the elements from the old memory block to the new one
  copy_range(this->elements, this->elements + count, ptrToNewBlock);

  // 3. Deallocate the old memory block
  delete[] this->elements;          // frees memory from old elements
  this->elements = ptrToNewBlock;   // redirects pointer to new block

  // 4. Add the new element at the end of the new memory block
  this->elements[count] = new_element;
  count++;                          // increment counter
}
```

Fragen/Unklarheiten?

6. Memory Management

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 }
```

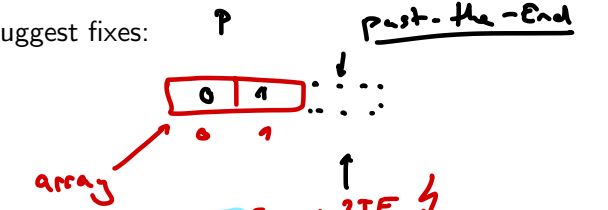
Find mistakes in the following code and suggest fixes:

```
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:

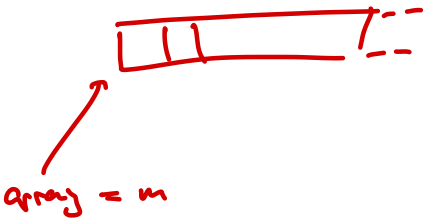
```
1  // PRE: len >= 2
2  int* fib(unsigned int len) {
3      int* array = new int[len];
4      array[0] = 0; array[1] = 1;
5      for (int* p = array+2; p < array + len; ++p) {
6          *p = *(p-2) + *(p-1); }
7      return array; }
8  void print(int* array, int len) {
9      for (int* p = array+2; 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 }
```

Fragen/Unklarheiten?

7. Outro

Allgemeine Fragen?

Bis zum nächsten Mal

Schöne Woche!