Exercise Session — Computer Science — 12

Adel Gavranović

Pointer-Arithmetik, Speichermanagement

Übersicht

- Follow-up
- Pointers
  - Example: Pointers on Arrays
  - Example: Special Copy
- Aufgabe "Push Back"
- Memory Management
- Muddiest Point

n.ethz.ch/~agavranovic

1. Intro

Intro

- Any new people here today?

2. Follow-up

Muddiest Point

Muddiest Point

Schauen wir uns am Ende der Session nochmal kurz an

Shared Pointers

Shared Pointers

- Das Code-Beispiel "Shared Pointers (explanation)" illustriert die Verwendung und den Nutzen von Shared Pointern ziemlich gut
- Falls doch jemand fragen hat, könnt ihr sie jetzt stellen

Fragen/Unklarheiten?

3. Lernziele

Ziele

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

4. Zusammenfassung

5. Pointers

`new` VS `new[]`

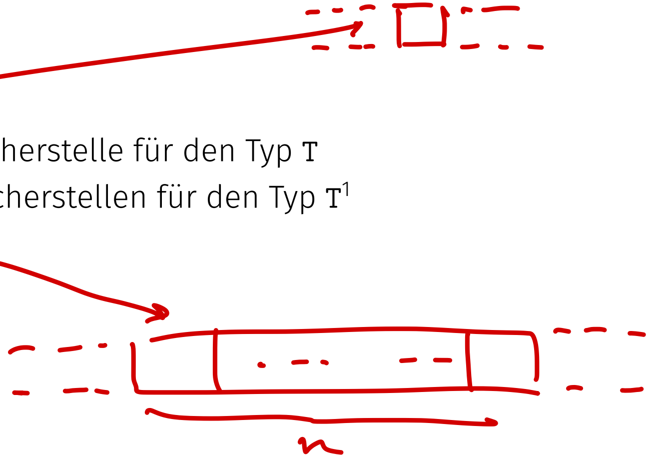
return pointer to newly alloc. object

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

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

`new` vs `new[]`

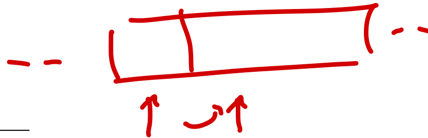
- `new T` alloziert **eine** Speicherstelle für den Typ `T`
- `new T[n]` alloziert **n** Speicherstellen für den Typ `T`[1]



[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



[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`

read & write

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

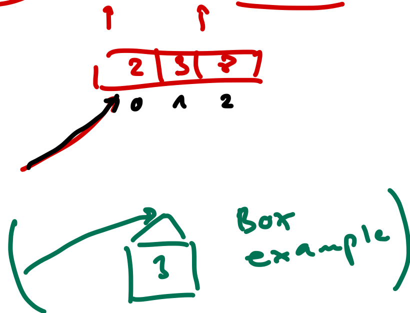
Statisch allozierter Array

```
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

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 alloziertere 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

Side Note: Arrays are weeeeird

Side Note: Arrays are weeeeird

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

std::cout << &myStatArr[0]  << "\t &myStatArr[0]  \n";
std::cout << myStatArr      << "\t myStatArr      \n";
std::cout << &myStatArr     << "\t &myStatArr     \n";
std::cout << &myDynArr[0]   << "\t &myDynArr[0]   \n";
std::cout << myDynArr       << "\t myDynArr       \n";
std::cout << &myDynArr      << "\t &myDynArr      \n\n\n";
std::cout << typeid(&myStatArr[0]).name() << "\tType of &myStatArr[0]\n";
std::cout << typeid(myStatArr).name()     << "\tType of myStatArr\n";
std::cout << typeid(&myStatArr).name()    << "\tType of &myStatArr\n";
std::cout << typeid(&myDynArr[0]).name()  << "\tType of &myDynArr[0]\n";
std::cout << typeid(myDynArr).name()      << "\tType of myDynArr\n";
std::cout << typeid(&myDynArr).name()     << "\tType of &myDynArr\n";
```

Side Note: Arrays are weeeeird

Side Note: Arrays are weeeeird

```
0x7ffcb4fe1d14   &myStatArr[0]
0x7ffcb4fe1d14   myStatArr
0x7ffcb4fe1d14   &myStatArr
0x2340fc0        &myDynArr[0]
0x2340fc0        myDynArr
0x7ffcb4fe1d08   &myDynArr


Pi       Type of &myStatArr[0]
A3_i     Type of myStatArr
PA3_i    Type of &myStatArr
Pi       Type of &myDynArr[0]
Pi       Type of myDynArr
PPi      Type of &myDynArr
```

`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

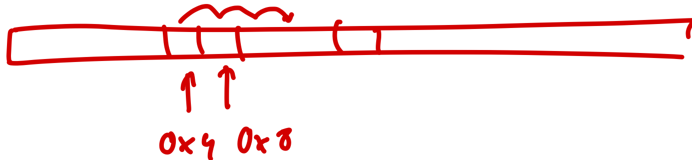
- Wir können mit Pointern "rechnen"
- Die wichtigsten Befehle sind:

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

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

- Temporäre Shifts

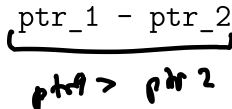
```
ptr + 3
ptr - 3
```

- Permanente Shifts

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

- Distanz zwischen Pointern bestimmen

```
ptr_1 - ptr_2
```

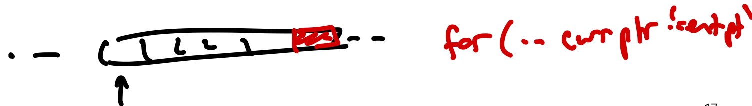


Pointer-Arithmetik

- 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
  ```
- Positionen vergleichen
  ```
  ptr_1 < ptr_2
  ptr_1 != ptr_2
  ```



Fragen/Unklarheiten?

5.1. Example: Pointers on Arrays

Pointer-Arithmetik

```
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
  // ^ Note how this does not simply "add 2" to the
  // underlying memory address, but instead adds the
  // appropriate amount to get to the integer variable
  // that is stored "2 ints further away"
*ptr = 18;                                      // overwrite target
int* past = a+5;
std::cout << (ptr < past) << "\n";              // compare pointers
```

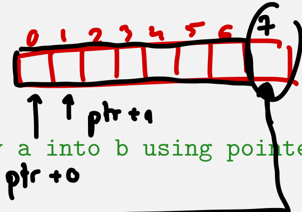
Bug Hunt

Finde und behebe mindestens 3 Probleme in folgendem Programm

```
int* a = new int[7]{0, 6, 5, 3, 2, 4, 1};
int* b = new int[7];
int* c = b;

for (int* p = a; p <= a+7; ++p) {   // copy a into b using pointers
    *c++ = p;
}

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



Bug Hunt

Finde und behebe mindestens 3 Probleme in folgendem Programm

```
int* a = new int[7]{0, 6, 5, 3, 2, 4, 1};
int* b = new int[7];
int* c = b;

for (int* p = a; p <= a+7; ++p) {    // copy a into b using pointers
    *c++ = *p;
}

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

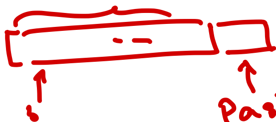
Probleme: `p`, `i` werden bei `a+7` dereferenziert; `c` zeigt nicht mehr auf `b[0]`!

Fragen/Unklarheiten?

5.2. Example: Special Copy

Special Copy?

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

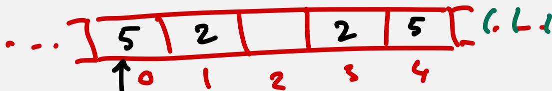


Reverse Copy!

```
// 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;
    }
}
```

Reverse Copy!

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

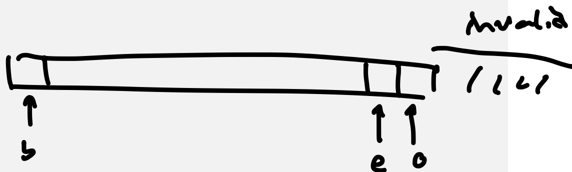


Welche dieser Eingaben sind valid nach `int* a = new int[5];`?

a) `f(a, a+5, a+5)` b) `f(a, a+2, a+3)` c) `f(a, a+3, a+2)`

Reverse Copy!

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



Welche dieser Eingaben sind valid nach `int* a = new int[5];`?

a) `f(a, a+5, a+5)` b) `f(a, a+2, a+3)` c) `f(a, a+3, a+2)`

Antwort: b)

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 `a`

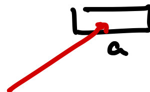
Pointer Costness

Es gibt zwei Arten von Constness bei Pointern:

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

kein Schreibzugriff auf `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;
```

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

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

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

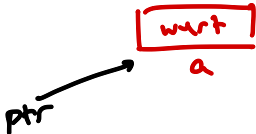
kein Schreibzugriff auf `ptr`

Pointer Costness

Es gibt zwei Arten von Constness bei Pointern:

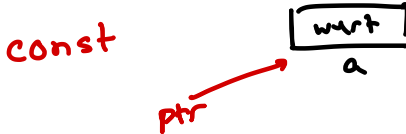
```
const int* ptr = &a;
```

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




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

kein Schreibzugriff auf `ptr`
d.h. wir dürfern nicht ändern, wohin der Pointer zeigt



Fragen/Unklarheiten?

6. 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;
//       source_ends points to element after the last element to be copied;
//       destination_begin points to first element of target memory block;
//       #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?

7. Memory Management

Bug Hunt I

Follow-up

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

Finde Fehler im Code und schlage Korrekturen vor

Bug Hunt I — Dangerous Pointer

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

Auch wenn der Pointer nicht dereferenziert wird, muss er in einen Speicherblock oder auf das Element unmittelbar nach dessen Ende zeigen.

Bug Hunt II

```
// PRE: len >= 2
int* fib(int len) {
    int* array = new int[len];
    array[0] = 0; array[1] = 1;
    for (int* p = array+2; p < array + len; ++p) {
        *p = *(p-2) + *(p-1); }
    return array; }
void print(int* array, int len) {
    for (int* p = array+2; p < array + len; ++p) {
        std::cout << *p << " ";
    }
}
void test2(int len) {
    int* array = fib(len);
    print(array, len);
}
```

Bug Hunt II — Memory Leak

```
// PRE: len >= 2
int* fib(int len) {
    int* array = new int[len];
    array[0] = 0; array[1] = 1;
    for (int* p = array+2; p < array + len; ++p) {
        *p = *(p-2) + *(p-1); }
    return array; }
void print(int* array, int len) {
    for (int* p = array+2; p < array + len; ++p) {
        std::cout << *p << " ";
    }
}
void test2(int len) {
    int* array = fib(len);
    print(array, len);
    delete[] array;          // otherwise array is leaked!
}
```

Bug Hunt III

```
// PRE: len >= 2
int* fib(int len) {
    // ...
}
void print(int* m, int len) {
    for (int* p = m+2; p < m + len; ++p) {
        std::cout << *p << " ";
    }
    delete m;
}
void test2(int len) {
    int* array = fib(len);
    print(array, len);
    delete[] array;
}
```

Bug Hunt III — Double Free!

```cpp
// PRE: len >= 2
int* fib(int len) {
    // ...
}
void print(int* m, int len) {
    for (int* p = m+2; p < m + len; ++p) {
        std::cout << *p << " ";
    }
    delete[] m;
}
void test2(int len) {
    int* array = fib(len);
    print(array, len);
    // delete[] array;          // array deallocated twice!
}
```

Fragen/Unklarheiten?

8. Muddiest Point

Woran hakt es bei dir?

(Absichtlich leer gelassen)

(Absichtlich leer gelassen)

(Absichtlich leer gelassen)

(Absichtlich leer gelassen)

(Absichtlich leer gelassen)

(Absichtlich leer gelassen)

9. Outro

Allgemeine Fragen?

Bis zum nächsten Mal

Schöne Woche noch!