

Übungsstunde

Woche 13

Adel Gavranović

`adel.gavranovic@inf.ethz.ch`

Übersicht

Heutige Themen

Intro

Self-Assessment

-tors

Exercise "Box"

Vokabular

Bug Hunt

Outro

Links

▶ [polybox zum Material für die Übungsstunden](#)

▶ [Mail an Assistenten](#)

Kommentare zu [code] expert

E11:T1 Understanding struct & classes

- POST
- Bei praktisch allen war die POST zwar faktisch korrekt, aber halt unnötig und nicht auf den Kontext angepasst
- POST sollen vermitteln was die Funktion *in diesem Kontext* macht und nicht einfach den darunterliegenden Code in Worte fassen
- Vergesst nicht, Expressions immer Schicht für Schicht zu evaluieren (z.B. `str.b == vec[1]`)

E11:T2 Averager

- Alle alles gut gemacht

Kommentare zu [code] expert

E11:T3a Understanding Pointers: - Lookup

- Spart euch die Zwischenvariablen

```
1 // instead of
2 const int* i_ptr = &vec.at(i);
3 return i_ptr;
4
5 // just do
6 return &vec.at(i);
```

E11:T3b Understanding Pointers - Add

- Ebenfalls alle alles gut gemacht

Kommentare zu [code] expert

E11:T3c Understanding Pointers - Num_Enum

- Kein Grund eine `for`-loop einzubauen. Wieso?
- Weil *Pointer Arithmetic* die gesamte Berechnung für uns erledigt

E11:T3d Understanding Pointers - First_Char

- Werte, die passed by value wurden, kann/soll man direkt "verwenden" (increment, decrement, etc.). Hier gibt es auch keinen Grund eine Zwischenvariable zu init'en

E11:T4 Quick Sort

- FYI: "temporary variables" nennen wir oft einfach `tmp`
- Studiert die Task Descriptions **und befolgt sie genau**
- Ich möchte wirklich nicht, dass jemand von euch die Prüfung wegen so Zeug versemzelt...

Fragen zu [code]expert eurerseits?

Lernziele

- Code tracen können, der `new`, `delete`, copy-Konstruktoren und Destruktoren beinhaltet
- Datenstrukturen¹ implementieren können, die wie primitive Typen agieren, aber intern mit dynamic memory implementiert wurden
- Häufige Probleme und Bugs beim Umgang mit Pointers kennen und vorbeugen können, insb.
 - dangling pointers
 - double-free
 - use-after-free
- Unterschied zwischen `new` bzw. `delete` und `new []` bzw. `delete []` kennen und wissen, wann welches Keyword zum Einsatz kommt

¹basically fancy types

Self-Assessment IV

Neues Self-Assessment (IV) ist freigeschaltet.

Macht es als Hausaufgabe und nehmt euch genug Zeit die Musterlösung zu studieren

Da war doch was...

Vergesst nicht...

Zu jedem `new` ein `delete` .

Constructor, Copy-Constructor, Destructor

- Sind einfach Funktionen, die zu bestimmten Anlässen gecalled werden
- Müssen `public` sein

Constructor

Constructor

- wird gerufen, wenn ein Objekt einer class/struct constructed wird
- Man kann dem Constructor Argumente geben, um das Objekt genau so zu initialisieren wie wir wollen
- Man kann mehrere Constructoren haben, z.B. für verschiedene Typen. Der Computer schliesst dann auf den richtigen Typ. Beispiel:

```
Personclass Person1(142.0f) oder
```

```
Personclass Person161('A')
```

- Mehr dazu: [▶ cppreference link](#)

Constructor Beispiel in einer class

Eine Möglichkeit eine Class mit Constructor zu definieren

```

1   class classname {
2       int a, b;
3   public:
4       const int& r;
5
6       classname(int i)
7           : r(a)    // initializes X::r to refer to X::a
8             , a(i)  // initializes X::a to the value of i
9             , b(i+5) // initializes X::b to the value of i+5
10          // ^^^^^ here we are using a "member initializer list"
11          { } // <- if you want your constructor to do anything
12            //      additionally, put it in there
13      };

```

Member Initializer List

```
1     Klassenname::Klassenname() : Membervariable_1(0) {  
2         Membervariable_2 = 0;  
3     }
```

Was ist der Unterschied zwischen diesen beiden Initialisierungen der Membervariablen? Wieso machen wir uns die Mühe MILs zu verwenden?

const Objekte

- In manchen Fällen möchte man `const` Member haben und die zweite Option würde dort nicht funktionieren

Performance

- Der Hauptgrund für uns ist aber eigentlich Performance. Der Code mit MILs ist schneller, da er unnötige Kopien vermeidet. Diese Kopien sehen wir nicht im Code, aber die Laufzeit/Performance wird dadurch schlechter

► gutes Video dazu

Destructor

Destructor

- wird gerufen, wenn ein Objekt einer class/struct *deconstructed* wird. Das kann passieren, am Ende eines Scopes oder wenn `delete` verwendet wird
- Wird genutzt, um Memory "sauber" zu halten, wenn ein Objekt nicht länger verwendet wird

Destructor example in a class

Eine Möglichkeit eine Class mit Deconstructor zu definieren

```
1   class classname {
2       int* value;
3   public:
4
5       // oter -cors and stuff go here
6
7       ~classname(){
8           delete value;
9           // that's how we clean up the value
10          // which lies at the slot that
11          // the int-pointer is pointing to,
12          // instead of just deleting the int-pointer
13          // (avoiding "memory leaks")
14      }
15 };
```

Copy-constructor

Copy-Constructor

- wird gerufen, wenn ein Objekt mit einem anderen Objekt der selben class/struct *initialisiert* wird
- es gibt ein default copy-constructor, *falls* wir keinen explizit deklarieren. Dieser macht einfach eine member-wise copy der class/struct
- lässt uns präzise bestimmen, wie wir etwas kopieren möchten statt einfach eine *shallow copy* zu machen
- nicht zu verwechseln mit dem `operator=`, der etwas sehr ähnliches macht

Shallow Copy

Was der default copy-constructor macht.

Wir wollen meistens eine *deep copy*.

Assignment-operator (=)

Assignment-operator (=)

- wird gerufen, wenn ein Objekt einem anderen Objekt der selben class/struct *assigned* wird
- wird *nur nach* (nicht bei) Initialisierungen gerufen
- heisst "assignment operator", so wie bei primitiven Types (=)
- Faustregel: macht erst destructor-Zeugs, dann copy-constructor-Zeugs
- *muss* einen return type haben, meist `classname&` damit man *chained assignments* machen kann (`a = b = c = d;` , allen wird `d` assigned)

operator= vs. Copy-Constructor

```
1 // our class/struct is named "Box"
2
3 Box first;           // init by default constructor
4 Box second(first);  // init by copy-constructor
5 Box third = first;  // also init by-copy constructor!
6 second = third;     // assignment by copy-assignment operator
```

Fragen/Unklarheiten?

Exercise "Box (copy)"

Hier schauen wir uns die Implementation *sehr* genau an.

Members of "Box"²

```
1  Box::Box(const Box& other) {
2      ptr = new int(*other.ptr);
3  }
4
5  Box& Box::operator= (const Box& other) {
6      *ptr = *other.ptr;
7      return *this;
8  }
```

²with all `std::cerr` removed

Members of "Box"³

```
1  Box::~~Box() {
2      delete ptr;
3      ptr = nullptr;
4  }
5
6  Box::Box(int* v) {
7      ptr = v;
8  }
9
10 int& Box::value() {
11     return *ptr;
12 }
```

³with all `std::cerr` removed

Tracing test_destructor1()

```
1 void test_destructor1() {
2     std::cerr << "[enter] test_destructor1" << std::endl;
3     int a;
4     {
5         Box box(new int(1));
6         a = 5;
7     }
8     std::cout << "a = " << a << std::endl;
9     std::cerr << "[exit] test_destructor1" << std::endl;
10 }
```

Tracing test_destructor2()

```
1 void test_destructor2() {
2     std::cerr << "[enter] test_destructor2" << std::endl;
3     {
4         Box* box_ptr = new Box(new int(2));
5         delete box_ptr;
6     }
7     std::cerr << "[exit] test_destructor2" << std::endl;
8 }
```


Tracing test_copy_constructor()

```
1 void test_copy_constructor() {
2     std::cerr << "[enter] test_copy_constructor" << std::endl;
3     {
4         Box demo(new int(0));
5         Box demo_copy = demo;
6         // assert(demo.value() == 0);
7         // assert(demo_copy.value() == 0);
8         demo.value() = 4;
9         // assert(demo.value() == 4);
10        // assert(demo_copy.value() == 0);
11        demo_copy.value() = 5;
12        // assert(demo.value() == 4);
13        // assert(demo_copy.value() == 5);
14    }
15    std::cerr << "[exit] test_copy_constructor" << std::endl;
16 }
```

Tracing test_copy_constructor()

Tracing test_assignment()

```
1 void test_assignment() {
2     std::cerr << "[enter] test_assignment" << std::endl;
3     {
4         Box demo(new int(0));
5         demo.value() = 3;
6         Box demo_copy(new int(0));
7         demo_copy = demo;
8         // assert(demo.value() == 3);
9         // assert(demo_copy.value() == 3);
10        demo.value() = 4;
11        // assert(demo.value() == 4);
12        // assert(demo_copy.value() == 3);
13        demo_copy.value() = 5;
14        // assert(demo.value() == 4);
15        // assert(demo_copy.value() == 5);
16    }
17    std::cerr << "[exit] test_assignment" << std::endl;
18 }
```

Tracing test_assignment()

Fragen/Unklarheiten?

Dangling Pointer

Intro

○○○○○○

Self-Assessment

○

-tors

○○○○○○○○○○

Exercise "Box"

○○○○○○○○○○

Vokabular

●○○○○

Bug Hunt

○

Outro

○○○

Double-Free

Intro

○○○○○○

Self-Assessment

○

-tors

○○○○○○○○○○

Exercise "Box"

○○○○○○○○○○

Vokabular

○○●○○○

Bug Hunt

○

Outro

○○○

Use-after-Free

Intro

○○○○○○

Self-Assessment

○

-tors

○○○○○○○○○○

Exercise "Box"

○○○○○○○○○○

Vokabular

○○●○○

Bug Hunt

○

Outro

○○○

Memory Leak

[] or not to [] ?

Wann [] ?

Eigentlich ganz einfach:

- nutzt `new T[]` wenn ihr mehr als ein Objekt allozieren möchte
- nutzt `delete []` wenn ihr mehr als ein Objekt deallozieren möchte

Schaut euch unbedingt die [Summary 13](#) an!

Fragen/Unklarheiten?

Digital Entomology

- entspant euch
- atmet tief ein und aus
- findet die Bugs und kategorisiert sie

Informatik

Exercise Session 12



<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 }
```

Tipps für [code]expert

E13:T1 Operator delete

- basically wieder Bug Hunt

E13:T2 Array-based Vector, Rule of Three

- für sehr eleganten Code braucht ihr wahrscheinlich eine Hilfsfunktion, die ihr selbst implementieren müsst

E13:T3 Smart Pointers

- dito, die gleiche Funktion sogar
- vergesst nicht, immer schön zu überprüfen, ob ihr davor seid, einen `nullptr` zu dereferenzieren

Allgemeine Fragen?

Bis zum nächsten Mal

- macht eure Hausaufgaben
- bleibt gesund