Übungsstunde — Informatik — 11

Adel Gavranović

Memory Management, Probleme mit Pointer, Smart Pointer, Muddiest Point

Übersicht

Follow-up
Memory Management
Aufgabe "Box"
Häufige Probleme mit Pointer
Shared and Unique Pointers
Muddiest Point

n.ethz.ch/~agavranovic

Material

Webpage

Mail

1. Follow-up

Fragen/Unklarheiten?

2. Feedback zu **code** expert

3. Lernziele

Ziele

- ☐ Programme die `new`, `delete`, copy constructors und destructors verwenden tracen können
- ☐ Dynamischen Speicher verstehen
- ☐ Die Ursachen und Gefahren von dangling pointers, double-free und use-after-free kennen und verhindern können
- ☐ `shared_ptr` verwenden können

4. Zusammenfassung

5. Memory Management

`new` und `delete`

Vergesst nie…

Zu jedem `new` ein `delete`

`new` und `delete`

Vergesst nie…

Zu jedem `new` ein `delete`

Constructor, Copy-Constructor, Destructor

`new` und `delete`

Vergesst nie…

Zu jedem `new` ein `delete`

Constructor, Copy-Constructor, Destructor

- Sind einfach Funktionen, die zu bestimmten Anlässen gecalled werden

`new` und `delete`

Vergesst nie…

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

Constructor

Constructor

- Wird gerufen, wenn ein Objekt einer class/struct constructed wird

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

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 Person001(142.0f);`
  - `personClass Person161(420);`

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 Person001(142.0f);`
  - `personClass Person161(420);`
- Mehr dazu: cppreference link

Constructor - Beispiel in einer class

Constructor - Beispiel in einer class

```
class meineKlasse {
      int a, b;
  public:
      const int& r; // for reading only!

      // CONSTRUCTOR
      meineKlasse(int i)
        : a(i)      // initializes a to the value of i
        , b(i+5)    // initializes b to the value of i+5
        , r(a)      // initializes r to refer to a
        // ^  here we are using a "member initializer list"
        // and if you want your constructor to do
        // anything additionally, put it inside
        {/*here (like in a regular function!)*/}
};
```

Member Initializer List

```
meineKlasse::meineKlasse()
    : memberVariableEins(0)          // init memberVariableEins
    { memberVariableZwei = 0; }      // init memberVariableZwei
```

Was ist der Unterschied zwischen diesen beiden Initialisierungen der Membervariablen?

Member Initializer List

```
meineKlasse::meineKlasse()
    : memberVariableEins(0)          // init memberVariableEins
    { memberVariableZwei = 0; }      // init memberVariableZwei
```

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

Member Initializer List

```
meineKlasse::meineKlasse()
    : memberVariableEins(0)          // init memberVariableEins
    { memberVariableZwei = 0; }      // init memberVariableZwei
```

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

`const` members

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

Member Initializer List

```
meineKlasse::meineKlasse()
    : memberVariableEins(0)          // init memberVariableEins
    { memberVariableZwei = 0; }      // init memberVariableZwei
```

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

`const` members

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

Performance

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

gutes Video dazu

Destructor

Destructor

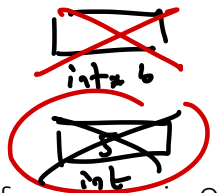
- wird gerufen, wenn

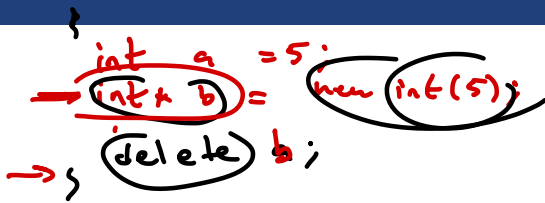
Destructor

Destructor

- wird gerufen, wenn ein Objekt einer class/struct *de*structed wird. Das kann passieren,

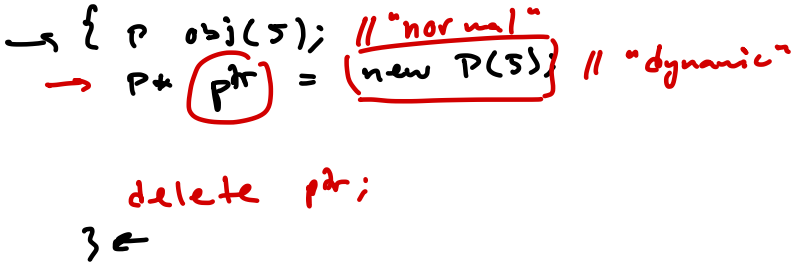
Destructor





Destructor

- wird gerufen, wenn ein Objekt einer class/struct *de*structed wird. Das kann passieren, am Ende eines Scopes oder wenn `delete` verwendet wird



Destructor

Destructor

- wird gerufen, wenn ein Objekt einer class/struct *de*structed 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 - Beispiel in einer class

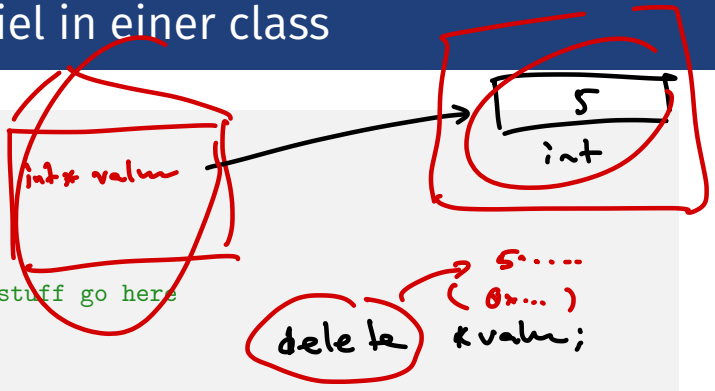
Destructor - Beispiel in einer class

```
class meineKlasse {
    int* value;

public:

    // other -ctors and stuff go here

    ~meineKlasse(){
        *value = 7;
        delete value;   // That's how we clean up the value which
                        // lies at the slot that the int-pointer is
                        // pointing to, instead of just deleting
                        // the int-pointer (avoiding "memory leaks")
    }
};
```



delete erwartet ein Adresse (pointer!!!)

Copy-constructor

Copy-Constructor

- wird gerufen, wenn

Copy-constructor

Copy-Constructor

- wird gerufen, wenn ein Objekt mit einem anderen Objekt derselben class/struct *initialisiert* wird

Copy-constructor

Copy-Constructor

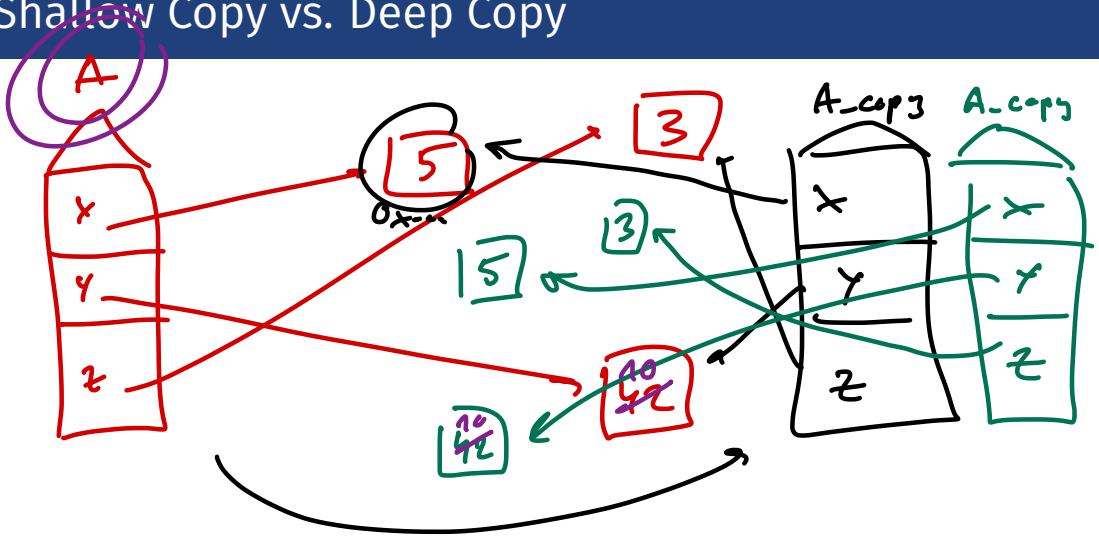
- wird gerufen, wenn ein Objekt mit einem anderen Objekt derselben class/struct *initialisiert* wird
- es gibt ein default copy-constructor, *falls* wir keinen explizit deklarieren. Dieser macht einfach eine member-wise copy der class/struct
- lässt uns präzise bestimmen, wie wir etwas kopieren möchten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

Copy-constructor

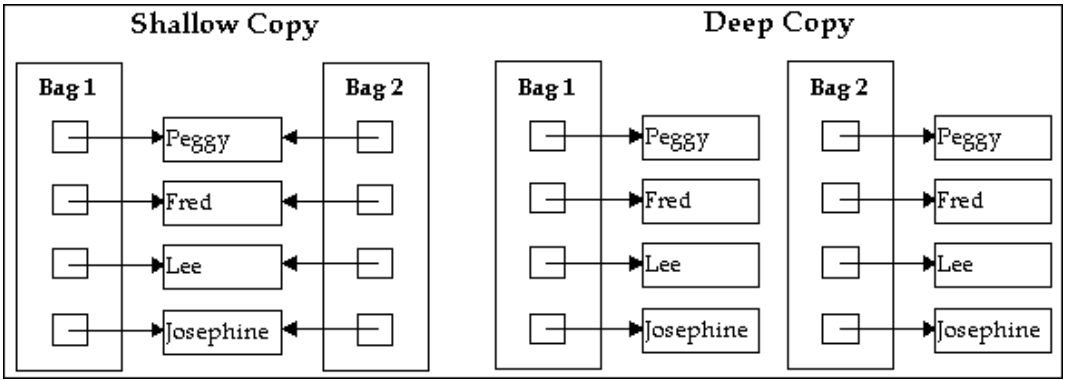
Copy-Constructor

- wird gerufen, wenn ein Objekt mit einem anderen Objekt derselben 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 vs. Deep Copy



Shallow Copy vs. Deep Copy



(copy-)assignment-operator (=)

Assignment-operator (=)

- wird gerufen, wenn

(copy-)assignment-operator (=)

Assignment-operator (=)

- wird gerufen, wenn ein Objekt einem anderen Objekt derselben class/struct *assigned* wird

(copy-)assignment-operator (=)

Assignment-operator (=)

- wird gerufen, wenn ein Objekt einem anderen Objekt derselben class/struct *assigned* wird
- wird *nur nach* (nicht bei) Initialisierungen gerufen

(copy-)assignment-operator (=)

Assignment-operator (=)

- wird gerufen, wenn ein Objekt einem anderen Objekt derselben class/struct *assigned* wird
- wird *nur nach* (nicht bei) Initialisierungen gerufen
- heisst "assignment operator", so wie bei primitiven Types (=)

(copy-)assignment-operator (=)

Assignment-operator (=)

- wird gerufen, wenn ein Objekt einem anderen Objekt derselben 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

(copy-)assignment-operator (=)

Assignment-operator (=)

- wird gerufen, wenn ein Objekt einem anderen Objekt derselben 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

(copy-)assignment-operator (=)

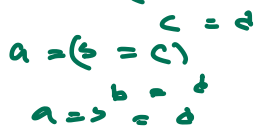
Assignment-operator (=)

- wird gerufen, wenn ein Objekt einem anderen Objekt derselben 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 `class&` damit

(copy-)assignment-operator (=)

Assignment-operator (=)

- wird gerufen, wenn ein Objekt einem anderen Objekt derselben 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 `class&` damit man *chained assigments* machen kann (`a = b = c = d;`, allen wird `d` assigned)



operator= vs. Copy-Constructor

```
// our class/struct is named "Box"

Box first;           // init by default constructor
Box second(first);   // init by copy-constructor
Box third = first;   // also init by copy-constructor
second = third;      // assignment by (copy-)assignment operator
```

operator= vs. Copy-Constructor

Info: slides are online now!

```
// our class/struct is named "Box"

Box first;           // init by default constructor
Box second(first);   // init by copy-constructor
Box third = first;   // also init by copy-constructor
second = third;      // assignment by (copy-)assignment operator
```

Die letzten beiden Fälle sehen ähnlich aus, aber denkt daran:
dass der (copy-)assignment-operator= erst in Aktion tritt, *nachdem* ein Objekt bereits initialisiert wurde

Fragen/Unklarheiten?

6. Aufgabe "Box"

Aufgabe "Box (copy)"

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

- Öffnet **code** expert und das Code-Beispiel "Box (copy)"

Aufgabe "Box (copy)"

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

- Öffnet **code** expert und das Code-Beispiel "Box (copy)"
- Macht euch noch keine Gedanken über `main.cpp`, dazu kommen wir noch

Aufgabe "Box (copy)"

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

- Öffnet **code** expert und das Code-Beispiel "Box (copy)"
- Macht euch noch keine Gedanken über `main.cpp`, dazu kommen wir noch
- Macht euch auch keine Gedanken über `std::cerr`, es ist nur ein fancy `std::cout`

Aufgabe "Box (copy)"

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

- Öffnet **code** expert und das Code-Beispiel "Box (copy)"
- Macht euch noch keine Gedanken über `main.cpp`, dazu kommen wir noch
- Macht euch auch keine Gedanken über `std::cerr`, es ist nur ein fancy `std::cout`
- Kleines code-together :)

Members von "Box"

```
Box::Box(const Box& other) {
    ptr = new int(*other.ptr);
}

Box& Box::operator= (const Box& other) {
    *ptr = *other.ptr;
    return *this;
}
```

Members von "Box"

```
Box::~Box() {
    delete ptr;
    ptr = nullptr;
}

Box::Box(int* v) {
    ptr = v;
}

int& Box::value() {
    return *ptr;
}
```

Tracing `test_destructor1()`

```
void test_destructor1() {
    std::cerr << "[enter] test_destructor1" << std::endl;

    int a;

    {
        Box box(new int(1));
        a = 5;
    }

    std::cout << "a = " << a << std::endl;
    std::cerr << "[exit] test_destructor1" << std::endl;
}
```

Tracing `test_destructor2()`

```
void test_destructor2() {
    std::cerr << "[enter] test_destructor2" << std::endl;

    {
        Box* box_ptr = new Box(new int(2));
        delete box_ptr;     // to trigger destructor of Box above
    }

    std::cerr << "[exit] test_destructor2" << std::endl;
}
```

Tracing `test_copy_constructor()`

```
void test_copy_constructor() {
    std::cerr << "[enter] test_copy_constructor" << std::endl;

    {
        Box demo(new int(0));
        Box demo_copy = demo;

        demo.value() = 4;

        demo_copy.value() = 5;
    }

    std::cerr << "[exit] test_copy_constructor" << std::endl;
}
```

Tracing `test_assignment()`

```
void test_assignment() {
    std::cerr << "[enter] test_assignment" << std::endl;

    {
        Box demo(new int(0));
        demo.value() = 3;
        Box demo_copy(new int(0));
        demo_copy = demo;
        demo.value() = 4;
        demo_copy.value() = 5;
    }

    std::cerr << "[exit] test_assignment" << std::endl;
}
```

Fragen/Unklarheiten?

7. Häufige Probleme mit Pointer

Dangling Pointers

Was?

[1] Oft ein *Zombie* gennant

Dangling Pointers

Was?

Ein *Dangling Pointer* ist ein Pointer der auf eine Speicherstelle zeigt, die bereits dealloziert wurde, also zeigt er auf eine invalide Speicherstelle.[1]

Wie?

[1] Oft ein *Zombie* gennant

Dangling Pointers

Was?

Ein *Dangling Pointer* ist ein Pointer der auf eine Speicherstelle zeigt, die bereits dealloziert wurde, also zeigt er auf eine invalide Speicherstelle.[1]

Wie?

Dies tritt häufig auf, wenn ein Objekt gelöscht wird oder aus dem Scope verschwindet, aber der Zeiger, der darauf zeigt, nicht auf `nullptr` gesetzt wird. So zeigt der Pointer immer noch auf den alten Speicherplatz, obwohl er nicht weiss, was sich jetzt dort befindet.

Na und?

[1] Oft ein *Zombie* gennant

Dangling Pointers

Was?

Ein *Dangling Pointer* ist ein Pointer der auf eine Speicherstelle zeigt, die bereits dealloziert wurde, also zeigt er auf eine invalide Speicherstelle.[1]

Wie?

Dies tritt häufig auf, wenn ein Objekt gelöscht wird oder aus dem Scope verschwindet, aber der Zeiger, der darauf zeigt, nicht auf `nullptr` gesetzt wird. So zeigt der Pointer immer noch auf den alten Speicherplatz, obwohl er nicht weiss, was sich jetzt dort befindet.

Na und?

Der Zugriff oder die Manipulation eines *Dangling Pointer* kann zu unvorhersehbarem Verhalten, Abstürzen oder Datenbeschädigung führen, da der Speicher möglicherweise neu zugewiesen wurde.

[1] Oft ein *Zombie* gennant

Double-Free

Was?

Double-Free

Was?

Double-free tritt auf, wenn `delete` zweimal für dieselbe Speicherzuweisung aufgerufen wird.

Wie?

Double-Free

Was?

Double-free tritt auf, wenn `delete` zweimal für dieselbe Speicherzuweisung aufgerufen wird.

Wie?

Dies tritt häufig in komplexen Programmen auf, bei denen die Speicherverwaltung an mehreren Stellen erfolgt, was zu Verwirrung darüber führt, wer über den Speicherplatz verfügt.

Na und?

Double-Free

Was?
Double-free tritt auf, wenn `delete` zweimal für dieselbe Speicherzuweisung aufgerufen wird.

Wie?
Dies tritt häufig in komplexen Programmen auf, bei denen die Speicherverwaltung an mehreren Stellen erfolgt, was zu Verwirrung darüber führt, wer über den Speicherplatz verfügt.

Na und?
Das doppelte Freigeben von Speicher kann die Speicherzuweisungsmetadaten beschädigen, was zu Speicherlecks, Programmabstürzen oder anderem fehlerhaften Verhalten führen kann.

Use-After-Free

Was?

Use-After-Free

Was?

Use-after-free ist eine Situation, in der ein Programm einen Zeiger weiter verwendet, nachdem es den Speicher, auf den er zeigt, freigegeben hat.

Wie?

Use-After-Free

Was?

Use-after-free ist eine Situation, in der ein Programm einen Zeiger weiter verwendet, nachdem es den Speicher, auf den er zeigt, freigegeben hat.

Wie?

Dies kann passieren, wenn das Programm den Zeiger nicht auf `nullptr` setzt, nachdem es ihn freigegeben hat, oder wenn es Kopien des Zeigers gibt, die nicht aktualisiert wurden.

Na und?

Use-After-Free

Was?

Use-after-free ist eine Situation, in der ein Programm einen Zeiger weiter verwendet, nachdem es den Speicher, auf den er zeigt, freigegeben hat.

Wie?

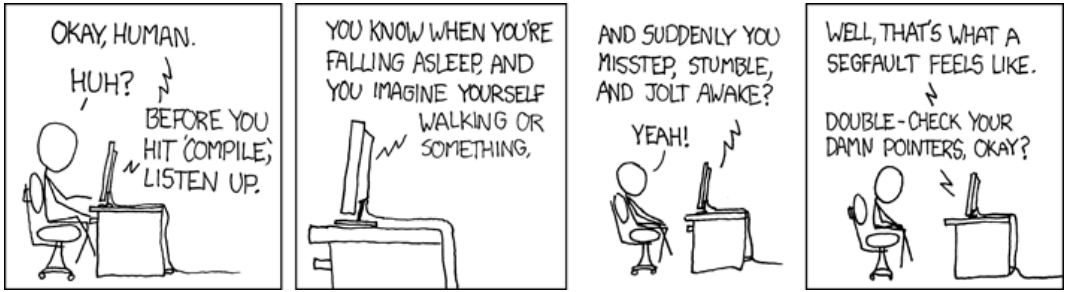
Dies kann passieren, wenn das Programm den Zeiger nicht auf `nullptr` setzt, nachdem es ihn freigegeben hat, oder wenn es Kopien des Zeigers gibt, die nicht aktualisiert wurden.

Na und?

Da der freigegebene Speicher möglicherweise für andere Zwecke neu zugewiesen wurde, kann seine Verwendung zu Datenbeschädigung, unvorhersehbarem Programmverhalten oder Sicherheitslücken führen.

*nullptr

*nullptr



xkcd

Fragen/Unklarheiten?

Dazu verdammt, Fehler zu machen?

Dazu verdammt, Fehler zu machen?

Wie kann man das alles verhindern?

Dazu verdammt, Fehler zu machen?

Wie kann man das alles verhindern?

Smart Pointers!

8. Shared and Unique Pointers

Smart Pointers

Smart Pointers

- Smart pointers sind bequeme Wrapper um reguläre Pointer, die helfen, Speicherlecks zu verhindern, indem sie den Speicher automatisch verwalten.
- Die Smart-Pointer `shared_ptr` und `unique_ptr` sind Teil der Standardbibliothek `<memory>`

Vergleich `unique_ptr` vs `shared_ptr`

`shared_ptr`

Vergleich `unique_ptr` vs `shared_ptr`

`shared_ptr`

Ein `shared_ptr` erlaubt es mehreren Zeigern, sich das Eigentum an derselben Ressource zu teilen. Es wird gezählt, wie viele Zeiger auf dieselbe Ressource zeigen. Sobald die Anzahl 0 erreicht, wird das Objekt gelöscht.

Vergleich `unique_ptr` vs `shared_ptr`

`shared_ptr`

Ein `shared_ptr` erlaubt es mehreren Zeigern, sich das Eigentum an derselben Ressource zu teilen. Es wird gezählt, wie viele Zeiger auf dieselbe Ressource zeigen. Sobald die Anzahl 0 erreicht, wird das Objekt gelöscht.

```
std::shared_ptr<SomeClass> s1 = std::make_shared<SomeClass>()
```

Vergleich `unique_ptr` vs `shared_ptr`

`shared_ptr`

Ein `shared_ptr` erlaubt es mehreren Zeigern, sich das Eigentum an derselben Ressource zu teilen. Es wird gezählt, wie viele Zeiger auf dieselbe Ressource zeigen. Sobald die Anzahl 0 erreicht, wird das Objekt gelöscht.

```
std::shared_ptr<SomeClass> s1 = std::make_shared<SomeClass>()
```

`unique_ptr`

Vergleich `unique_ptr` vs `shared_ptr`

`shared_ptr`

Ein `shared_ptr` erlaubt es mehreren Zeigern, sich das Eigentum an derselben Ressource zu teilen. Es wird gezählt, wie viele Zeiger auf dieselbe Ressource zeigen. Sobald die Anzahl 0 erreicht, wird das Objekt gelöscht.

```
std::shared_ptr<SomeClass> s1 = std::make_shared<SomeClass>()
```

`unique_ptr`

Ein `unique_ptr` wird für exklusives Eigentum verwendet. Speicher, der mit einem `unique_ptr` verbunden ist, wird automatisch freigegeben, wenn er den Scope verlässt.

Fragen/Unklarheiten?

9. Muddiest Point

Woran hakt es bei dir?

Q&A Session

10. Outro

Allgemeine Fragen?

Bis zum nächsten Mal

Follow-up:
- Q&A (muddiest point)
- shared ptr
- Snack (Samiklaus?)

Schöne Woche noch!