# Exercise Session — Computer Science — 11

Memory Management, Problems with Pointers, Shared Pointer Unique Pointer, Muddiest Point

# Overview

**Today's Plan**

Memory Management
Exercise "Box"
Common Issues with Pointers
Shared and Unique Pointers
Muddiest Point

n.ethz.ch/~iopopa

Link to Webpage

Send an e-Mail

# 1. Memory Management

# `new` and `delete`

**Never forget…**

For each **`new`** a **`delete`**

# `new` and `delete`

**Never forget…**

For each **`new`** a **`delete`**

**Constructor, Copy-Constructor, Destructor**

# `new` and `delete`

**Never forget…**

For each `new` a `delete`

**Constructor, Copy-Constructor, Destructor**

- Are just functions which are called at certain events

# `new` and `delete`

**Never forget...**

For each `new` a `delete`

**Constructor, Copy-Constructor, Destructor**

- Are just functions which are called at certain events
- Must be `public`

# Constructor

**Constructor**

- Called when

## Constructor

- Called when an object of a class/struct is constructed

# Constructor

**Constructor**

- Called when an object of a class/struct is constructed
- We can give the constructor arguments in order to initialize the object as we want

# Constructor

## Constructor

- Called when an object of a class/struct is constructed
- We can give the constructor arguments in order to initialize the object as we want
- There can be multiple constructors, e.g. for different types. The computer then infers the correct type. For example:
  - `personClass Person001(142.0f);`
  - `personClass Person161(45);`

## Constructor

- Called when an object of a class/struct is constructed
- We can give the constructor arguments in order to initialize the object as we want
- There can be multiple constructors, e.g. for different types. The computer then infers the correct type. For example:
  - `personClass Person001(142.0f);`
  - `personClass Person161(45);`
- More on this: cppreference link

# Constructor - Example in a class

# Constructor - Example in a class

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

      // CONSTRUCTOR
      meineKlasse(int i)
        : a(i)      // initializes r to refer to a
        , b(i+5)    // initializes a to the value of i
        , r(a)      // initializes b to the value of i+5
        // ^  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
```

What is the difference between these two initializations of the member variables?

# Member Initializer List

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

What is the difference between these two initializations of the member variables? Why do we use MILs?

# Member Initializer List

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

What is the difference between these two initializations of the member variables? Why do we use MILs?

**`const` members**

- In some cases we want to have `const` members and the second option would not work

# Member Initializer List

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

What is the difference between these two initializations of the member variables? Why do we use MILs?

**`const` members**

- In some cases we want to have `const` members and the second option would not work

**Performance**

- The main reason for us is performance. The code with MILs is faster, as it avoids unnecessary copies. We do not see these copies in the code but they worsen the runtime/performance good video on this

# Destructor

**Destructor**

- is called when

# Destructor

**Destructor**

- is called when an object of a class/struct is *de*constructed. This can happen

# Destructor

**Destructor**

- is called when an object of a class/struct is *de*constructed. This can happen at the end of a scope or when `delete` is used

# Destructor

**Destructor**

- is called when an object of a class/struct is *de*constructed. This can happen at the end of a scope or when `delete` is used
- is used to keep memory "clean" when an object is no longer in use

# Destructor - Example in a class

# Destructor - Example in a class

```
class meineKlasse {
    int* value;

public:

    // other -ctors and stuff go here

    ~meineKlasse(){

        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")
    }

};
```

# Copy-constructor

**Copy-Constructor**

- is called when

# Copy-constructor

**Copy-Constructor**

- is called when an object is *initialized* with another object of the same class/struct

# Copy-constructor

**Copy-Constructor**

- is called when an object is *initialized* with another object of the same class/struct
- there is a default copy constructor, *if* we don't declare one explicitly. This simply makes a member-wise copy of the class/struct
- lets us precisely determine how we want to copy something instead of simply doing a *shallow copy*

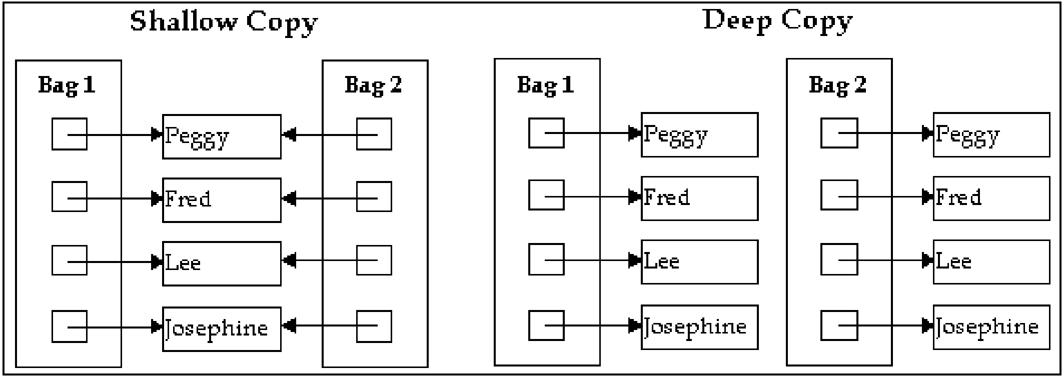
# Copy-constructor

## Copy-Constructor

- is called when an object is *initialized* with another object of the same class/struct
- there is a default copy constructor, *if* we don't declare one explicitly. This simply makes a member-wise copy of the class/struct
- lets us precisely determine how we want to copy something instead of simply doing a *shallow copy*
- not to be confused with the `operator=`, which does something very similar

# Shallow Copy vs. Deep Copy

# Shallow Copy vs. Deep Copy



# (copy-)assignment-operator (=)

**Assignment-operator (=)**

- is called when

# (copy-)assignment-operator (=)

**Assignment-operator (=)**

- is called when an object is *assigned* to another object of the same class/struct

# (copy-)assignment-operator (=)

**Assignment-operator (=)**

- is called when an object is *assigned* to another object of the same class/struct
- is called *only after* (not during) initializations

# (copy-)assignment-operator (=)

**Assignment-operator (=)**

- is called when an object is *assigned* to another object of the same class/struct
- is called *only after* (not during) initializations
- is called "assignment operator", just as with primitive types

# (copy-)assignment-operator (=)

**Assignment-operator (=)**

- is called when an object is *assigned* to another object of the same class/struct
- is called *only after* (not during) initializations
- is called "assignment operator", just as with primitive types
- Rule of thumb: do destructor stuff first, then copy constructor stuff

# (copy-)assignment-operator (=)

**Assignment-operator (=)**

- is called when an object is *assigned* to another object of the same class/struct
- is called *only after* (not during) initializations
- is called "assignment operator", just as with primitive types
- Rule of thumb: do destructor stuff first, then copy constructor stuff
- *must* have a return type, usually

# (copy-)assignment-operator (=)

## Assignment-operator (=)

- is called when an object is *assigned* to another object of the same class/struct
- is called *only after* (not during) initializations
- is called "assignment operator", just as with primitive types
- Rule of thumb: do destructor stuff first, then copy constructor stuff
- *must* have a return type, usually `class&` so that

# (copy-)assignment-operator (=)

**Assignment-operator (=)**

- is called when an object is *assigned* to another object of the same class/struct
- is called *only after* (not during) initializations
- is called "assignment operator", just as with primitive types
- Rule of thumb: do destructor stuff first, then copy constructor stuff
- *must* have a return type, usually `class&` so that you can make *chained assigments* (`a = b = c = d;`, `d` is assigned to all)

# 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

```
// 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
```

The last two cases look similar, but remember:
the (copy-)assignment-operator= only comes into action *after* an object has already been initialized

# Questions?

# 2. Exercise "Box"

# Exercise "Box (copy)"

Here we'll take a *very* close look at the implementation

- Go to **code** expert and open the code example "Box (copy)"

# Exercise "Box (copy)"

Here we'll take a *very* close look at the implementation

- Go to **code** expert and open the code example "Box (copy)"
- Don't worry about `main.cpp` yet, we'll get to that

# Exercise "Box (copy)"

Here we'll take a *very* close look at the implementation

- Go to **code** expert and open the code example "Box (copy)"
- Don't worry about `main.cpp` yet, we'll get to that
- Don't worry about `std::cerr` either, it's just fancy `std::cout`

# Exercise "Box (copy)"

Here we'll take a *very* close look at the implementation

- Go to **code** expert and open the code example "Box (copy)"
- Don't worry about `main.cpp` yet, we'll get to that
- Don't worry about `std::cerr` either, it's just fancy `std::cout`
- Small code-together :)

# Members of "Box"

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

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

# Members of "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()

```cpp
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;
}
```

# Questions?

# 3. Common Issues with Pointers

# Dangling Pointers

**What?**

[1]Often referred to as a *Zombie*

# Dangling Pointers

**What?**

A *dangling pointer* arises when a pointer is pointing to a memory location that has been freed or deallocated. Essentially, the pointer is pointing to a place that is no longer valid.[1]

**How?**

[1] Often referred to as a *Zombie*

# Dangling Pointers

**What?**

A *dangling pointer* arises when a pointer is pointing to a memory location that has been freed or deallocated. Essentially, the pointer is pointing to a place that is no longer valid.[1]

**How?**

This often occurs when an object is deleted or goes out of scope, but the pointer pointing to it is not set to `nullptr`. As a result, the pointer still refers to the old memory location, despite not knowing what is there now.

**So?**

[1]Often referred to as a *Zombie*

# Dangling Pointers

**What?**
A *dangling pointer* arises when a pointer is pointing to a memory location that has been freed or deallocated. Essentially, the pointer is pointing to a place that is no longer valid.[1]

**How?**
This often occurs when an object is deleted or goes out of scope, but the pointer pointing to it is not set to `nullptr`. As a result, the pointer still refers to the old memory location, despite not knowing what is there now.

**So?**
Accessing or manipulating a *dangling pointer* can lead to unpredictable behavior, crashes, or data corruption, as the memory might be reallocated and used for something else.

[1]Often referred to as a *Zombie*

# Double-Free

**What?**

# Double-Free

**What?**

*Double-free* occurs when `delete` is called twice on the same memory allocation.

**How?**

# Double-Free

**What?**

*Double-free* occurs when `delete` is called twice on the same memory allocation.

**How?**

This often occurs in complex programs where memory management is handled in multiple places, leading to confusion about who owns the memory.

**So?**

# Double-Free

**What?**

*Double-free* occurs when `delete` is called twice on the same memory allocation.

**How?**

This often occurs in complex programs where memory management is handled in multiple places, leading to confusion about who owns the memory.

**So?**

Freeing memory twice can corrupt the memory allocation metadata, potentially leading to memory leaks, program crashes, or other erratic behavior.

# Use-After-Free

**What?**

# Use-After-Free

**What?**

*Use-after-free* is a situation where a program continues to use a pointer after it has freed the memory it points to.

**How?**

# Use-After-Free

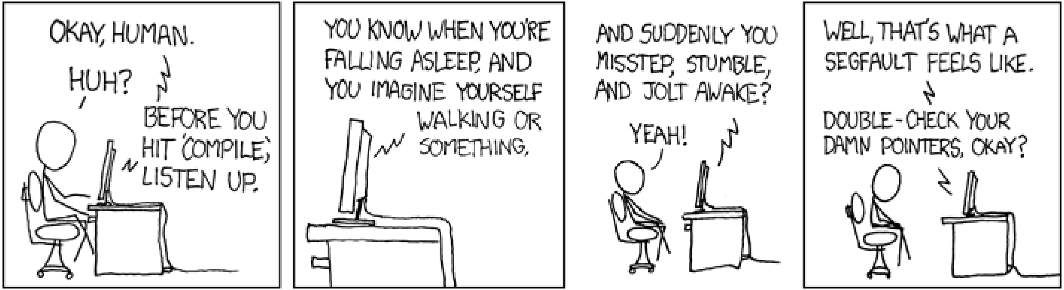
**What?**
*Use-after-free* is a situation where a program continues to use a pointer after it has freed the memory it points to.
**How?**
This can happen if the program does not set the pointer to `nullptr` after freeing it, or if there are copies of the pointer that were not updated.
**So?**

# Use-After-Free

**What?**
*Use-after-free* is a situation where a program continues to use a pointer after it has freed the memory it points to.
**How?**
This can happen if the program does not set the pointer to `nullptr` after freeing it, or if there are copies of the pointer that were not updated.
**So?**
Since the freed memory might be reallocated for other purposes, using it can lead to data corruption, unpredictable program behavior, or security vulnerabilities.

# *nullptr

# *nullptr




xkcd

# Questions?

# Doomed to cause errors?

# Doomed to cause errors?

How to prevent all this?

# Doomed to cause errors?

How to prevent all this?

**Smart Pointers!**

# 4. Shared and Unique Pointers

## Smart Pointers

- Smart pointers are convenient wrappers around regular pointers that help prevent memory leaks by automatically managing memory
- The smart pointers `shared_ptr` and `unique_ptr` are part of the standard `<memory>` library.

# Comparison unique_ptr vs shared_ptr

shared_ptr

# Comparison `unique_ptr` vs `shared_ptr`

**`shared_ptr`**
A `shared_ptr` allows multiple pointers to share ownership of the same resource. It counts how many pointers point to the same resource. Once the count reaches 0, the object is deleted.

# Comparison `unique_ptr` vs `shared_ptr`

`shared_ptr`

A `shared_ptr` allows multiple pointers to share ownership of the same resource. It counts how many pointers point to the same resource. Once the count reaches 0, the object is deleted.

`unique_ptr`

# Comparison `unique_ptr` vs `shared_ptr`

`shared_ptr`

A `shared_ptr` allows multiple pointers to share ownership of the same resource. It counts how many pointers point to the same resource. Once the count reaches 0, the object is deleted.

`unique_ptr`

A `unique_ptr` is used for exclusive ownership. Memory associated with a `unique_ptr` is automatically deallocated when they go out of scope.

# Questions?

# 5. Muddiest Point

# So, what are you stuck on?

**Q&A Session**

# 6. Outro

# General Questions?

# See you next time

Have a nice week!