ETH zürich

Exercise Session W12

Computer Science (CSE & CBB & Statistics) – AS 23

Overview

Today's Agenda

Follow-up
Feedback on **code** expert
Objectives
Memory Management
Exercise "Box"
Common Issues with Pointers
Shared and Unique Pointers
Muddiest Point
Outro

n.ethz.ch/~agavranovic

1. Follow-up

Follow-up from previous exercise sessions

- a visualized write-up for `our_list` is now avaiable on Lily's webpage!

`rwko.ch/lily`

2. Feedback on **code** expert

General things regarding **code** expert

- All of you have improved a lot - *well done!*

Specific things regarding **code** expert : `else`

Why do so almost all of you use format the **`else`** like this?

```
if(condition){
    something();
}

else {
    somehing_else();
}
```

or even *python*esque

```
if(condition){
    something();
else
    somehing_else();
```

(It's not wrong. Just *weird*)

Specific things regarding **code** expert : `else`

Consider formatting them like this

```
    if(condition){
        something();
    } else {
        somehing_else();
    }
```

Specific things regarding **code** expert : `else`

If your **`if`**-statement ends in a **`return`**, then you can leave out the **`else`** entirely! (This happens often with recursive functions!)

```
void recFoo(std::vector<unsigned int> numbers){
    // BASE CASE
    if(condition){
        return something();
    }

    // LONGER RECURSIVE CASE
    somehing_else0();
    somehing_else1();
    somehing_else2();
    somehing_else3();
    // ...
}
```

Specific things regarding **code** expert

E8:T2: "Recursive function analysis"

- the grading scheme is pretty strict on this one (one wrong PRE/POST means 1/3 points, only perfect solutions get 3/3)
- Always consider input 0
- some of you struggle with the formatting of TeX-stuff. always put math inside dollar signs on each end
- Check the master solution for a more standard phrasing for "termination proofs/arguments" (that is more likely to give points at the exam)
- The magic word in these "proofs" is "monotonically decreasing to the base case"
- Returning is not the same as printing

Specific things regarding **code** expert

E8:T3: "Bitstrings up to n"

- Don’t feel bad if you didn’t get the TA-points because of the computationally "terrible" implementation (the grading scheme forced me)
- I highly recommend studying the master solution since it’s very cool and concise and shows off a couple nice ways to use **`std::strings`**
- No recursion $\implies$ No TA-points (again, grading scheme forced me)

Specific things regarding **code** expert

E8:T4: "Trapezoid Printing"

- No recursion $\implies$ No TA-points (again, grading scheme forced me)
- Make sure to check the output of your code yourself and not rely fully on the autograder

Specific things regarding **code** expert

E9:T1: "Reverse Digits"

- You can output expressions directly instead of saving them in a variable first, i.e. instead of

```
int rest = n%10;
std::cout << rest;
int new_n = (n-rest)/10;
reverse(new_n);
```

- you can do the following:

```
std::cout << n%10;
reverse(n/10);
```

Questions?

3. Objectives

Objectives

- ☐ be able to trace code that uses `new`, `delete`, copy constructors, and destructors.
- ☐ understand the common problems related to incorrect use of dynamic memory: dangling pointers, double-free, use-after-free
- ☐ be able to define and use shared and unique pointers

4. Memory Management

`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 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

```
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
        // ^  4here 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? 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 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

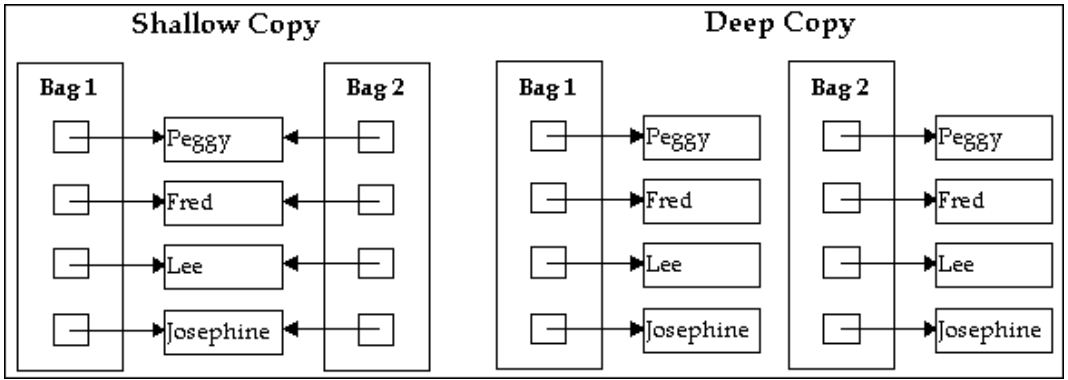
```
class meineKlasse {
    int* value;

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 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



(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
```

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

Questions?

5. 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)"
- 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`**
- Let's 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;
    }

    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_copy_constructor()`

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;
}
```

Tracing `test_assignment()`

Questions?

6. Common Issues with Pointers

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

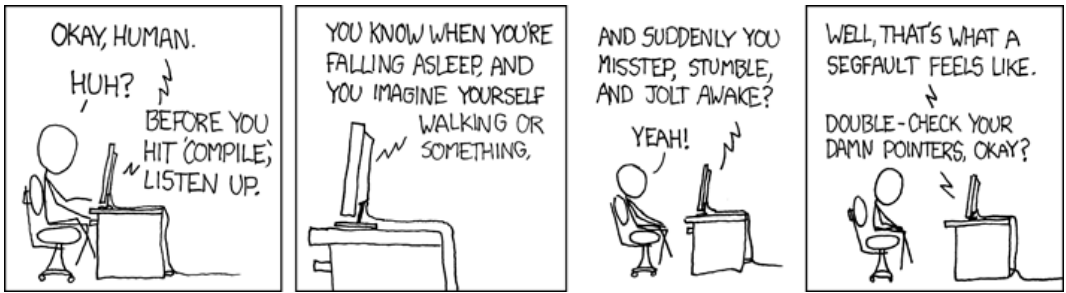
How?

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



▸ xkcd

Questions?

Doomed to cause errors?

How to prevent all this?

Smart Pointers!

7. Shared and Unique Pointers

Smart 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`

`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.

`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.

Smart Pointers in a nutshell

Questions?

8. Muddiest Point

So, what are you stuck on?

Q&A Session

9. Outro

General Questions?

Advertisement

Till next time!

Cheers!