Exercise Class 12

Objectives

The objectives of this exercise session are:

1. The students after the class should be able to trace code that uses `new`, `delete`, copy constructors, and destructors.
2. The students after the class should understand the common problems related to incorrect use of dynamic memory: dangling pointers, double-free, use-after-free.
3. The students after the class should be able to define and use shared pointers.

Prepare Before the Class

Contents

1 Homework Questions (< 10 min.)

Ask students if they have any questions about the homework exercises. If so, provide them support.

2 Memory Management (40 min.)

2.1 Exercise: Memory Management Issues (10 min.)

Before going to the memory management explanation, discuss common problems related to incorrect use of dynamic memory: dangling pointers, double-free, use-after-free.
Open the file `dangling.cpp` and follow these steps:

1. Comment out each part (there are 4 parts) and ask students to guess what will happen when they run the code.
2. Actually run the code and discuss the results, especially on why the code behaves the way it does.

NOTE: The `dangling.cpp` file needs to be run on your local machine and not on [code]expert, as the [code]expert environment prevents some of the issues from happening. Since some of the behavior may be platform-dependent, it is recommended to run the code on your local machine before the class. The code is also available in the `progs` folder.

The `dangling.cpp` file content for your reference:

```cpp
// dangling pointer/use-after-free/double-free examples

#include <iostream>

struct WeirdNumber {
    int number;

    void increment_by(int number){
        this->number += number;
    }
};

/* for part 4 - useful to also explain why this causes a
    compiler warning */
// int* weird_function() {
//     int x = 5;
//     return &x;
// }

int main() {

    /* part 1
    - run without memory checks, show the printed value
        likely changes every time, we get e.g.

        42
        -807906505

    */
```

```
    // WeirdNumber* p = new WeirdNumber {42};
    // std::cout << p->number << std::endl;

    // delete p;

    // p->increment_by(2);
    // std::cout << p->number << std::endl;

    /* part 2
    - illustrate how the dangling pointer can easily turn
         into a security vulnerability
      if we immediately allocate a new secret value, so the
           program prints

        42
        33

     */

    WeirdNumber* p = new WeirdNumber {42};
    std::cout << p->number << std::endl;

    delete p;
    WeirdNumber* secret = new WeirdNumber {31};

    p->increment_by(2);
    std::cout << p->number << std::endl;

    /* part 3
    - "fix" the above two issues by resetting p, but show
         what happens if we delete p twice */

    // WeirdNumber* p = new WeirdNumber {42};
    // std::cout << p->number << std::endl;

    // delete p;
    // // p = nullptr;

    // delete p;
    // std::cout << p->number << std::endl;

    /* part 4
    - show another way we can get a dangling pointer */

    // int* r = weird_function();
    // std::cout << *r << std::endl;

    return 0;
}
```

2.2 Explanation (30 min., [code]expert)

This lecture introduced the `delete` operator, copy constructors, and destructors.

Open "Box (copy)" task in [code]expert.

Destructors and `delete`.

1. Start the program and input `0` to run `test_destructor1`.

2. Show them how to trace `test_destructor1` by using the notation from the tracing handout `https://lec.inf.ethz.ch/ifmp/2024/guides/tracing/destructors.html`

   > The statement `a = 5;` is there only to have something between the constructor call and the destructor call so that it is easier to show that the destructor is called at the end of scope.

3. When you enter the destructor `Box::~Box()` ask the students:

   Question

   The purpose of the destructor is to perform clean-up. Based on the currently visualized state: what should the destructor do?

   Possible Answer

   Define a destructor in `box.cpp` that deletes the memory pointed at by `ptr`. Also, add a statement that outputs something with `cerr` at the end of destructor.

   ```
   Box::~Box() {
       std::cerr << "destructor: value=" << *ptr << " at "
             << ptr << std::endl;
       delete ptr;
       ptr = nullptr;
   }
   ```

4. Continue tracing until the end.

5.

   Question

   How can we know that our destructor is actually called?

   Possible Answer

   Run the program and show that the output from the destructor is printed in the console.

6. Run the program and input `1` to execute `test_destructor2`.

   Question

   Why don't we see the output from the destructor in this case?

   Answer

   Because the value to which `box_ptr` points is not deallocated automatically and thus leaked.

7. Fix the leak by adding a delete statement in `test_destructor2` of `main.cpp`:

```
delete box_ptr;
```

8. Rerun the program to show that the destructor is now called.

Questions?

Copy constructor.

1. Input `2` to run `test_copy_constructor`
2.

> **Question**
>
> Which method is called by this line:
>
> ```
> Box demo_copy = demo;
> ```
>
> **Answer**
> Copy constructor `Box::Box(const Box& other)`.

3. Define the following copy constructor:

```
Box::Box(const Box& other) {
  //'new int(v)' dynamically allocates a new int, and uses
      the
  //constructor of ints to initialise the new object.
  ptr = new int(*other.ptr);
  std::cerr << "copy constructor: value=" << *ptr << " at "
      << ptr << std::endl;
}
```

4. Run the program to show that the text from the copy constructor is printed.

Questions?

Assignment operator.

1. Input `3` to run `test_assignment`:
2.

> **Question**
>
> Which method is called by this line:
>
> ```
> demo_copy = demo;
> ```
>
> **Answer**
> Assignment operator:
> `Box& Box::operator= (const Box& other)`

3. Define the following assignment operator:

```
Box& Box::operator= (const Box& other) {
  *ptr = *other.ptr;
  std::cerr << "assignment operator: value=" << *ptr << " at
        " << ptr << std::endl;
  return *this;
}
```

4. Run the program to show that the text from the assignment operator is printed.

Questions?

3 Shared Pointers (10 min.)

> In case your students have a lot of questions about this topic, feel free to spend more time here and less time on the next section.

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

3.1 Explanation (10 min., [code]expert)

Open the "Shared Pointers (explanation)" task in [code]expert.

1.

> **Question**
>
> What are shared pointers?
>
> **Possible Answer**
> 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.

2. Start the program to run `shared_ptr_demo`. Go through the program and its output line by line.

> **Note:** If and only if students ask about unique pointers, this is a possible answer you could give them:
>
> A `unique_ptr` is used for exclusive ownership. Memory associated with a `unique_ptr` is automatically deallocated when they go out of scope.

4 Muddiest Point (15-30 min.)

At this point, the lecture has introduced several "hard" concepts like recursion, references/pointers and dynamic data structures. We would like to give the students the opportunity to think about these topics, identify what is not clear and ideally receive clarifications from you about these.
If you see that your students do not benefit from this activity or you need more time to prepare to explain the concepts, feel free to switch to the next exercises.

1. Ask the students to think for a couple of minutes which concepts are the most unclear to them. The students should write these concepts on a piece of paper or the blackboard during the break.
2. Ask students to vote which concepts are the most unclear.
3. Try to explain the most problematic concepts. If you see that you need to prepare more to be able to explain, please take a note and say to the students that we will give them additional explanations next week.

   Please note that the time you spend on explaining the identified problematic concepts is the time you could spend on explaining this week's topic. Therefore, please try to avoid spending time on things that are not common misunderstandings. For example, if you see that only a few students have problems with understanding differences between pointers and references, you could refer them to the study center.

   In case you see that almost all students have certain misunderstandings, please inform the head TA about them so that the teaching material can be improved to avoid them.
4. At this point, we expect recursion, pointers and data structures not to be fully understood but they should have the general concepts. If it is not the case, please repeat the theory before continuing with more exercises.

Questions?

5 Additional Material: Memory Management (15 min.)

The additional material sections provide some additional classroom activities which you can use in case you decide that the provided lesson plan does not work for you.

5.1 Exercise: Box (move) (15 min.)

5.1.1 Reading Task and Individual Thinking (10 min.)

1. Open the task "Box (move)" in [code]expert and show the task description to students.
2. Ask students to implement the task. Give them 10 minutes.

5.1.2 Share Solution (5 min., programming environment)

Ask students to dictate the solution; write it in the project. Point out and discuss any mistakes and inefficiencies.

The master solution for your reference:

```
Box::~Box() {
  if (!is_moved()) {
    std::cerr << "destructor: deallocating value=" << *ptr <<
          " at " << ptr << std::endl;
    delete ptr;
    ptr = nullptr;
  }
}

Box::Box(Box& other) {
  assert(!other.is_moved());
  ptr = other.ptr;
  other.ptr = nullptr;
  std::cerr << "copy constructor: value=" << *ptr << " at "
      << ptr << std::endl;
}

Box& Box::operator= (Box& other) {
  assert(!other.is_moved());
  if (this == &other) {
    // Assigning to itself, nothing to do.
    return *this;
  }
  if (!is_moved()) {
    std::cerr << "assignment operator: delete old value=" <<
        *ptr << " at " << ptr << std::endl;
    delete ptr;
    ptr = nullptr;
  }
  ptr = other.ptr;
  other.ptr = nullptr;
  std::cerr << "assignment operator: new value=" << *ptr << "
       at " << ptr << std::endl;
  return *this;
}
```

Questions?