ETH zürich

Exercise Session W11

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

Overview

`n.ethz.ch/~agavranovic`

Today's Agenda

Follow-up
Feedback on **code** expert
Objectives
`&` vs `*`
References vs Pointers
`this->`
Dynamic Data Structures & Iterators
Outro

1. Follow-up

Follow-up from previous exercise sessions

Regarding PVK

- By now you should all have received an e-mail from VMP informing you about the PVKs

2. Feedback on **code** expert

Specific things regarding **code** expert tasks

E8:T1: "Vector and Matrix Operations"

- Pay attention to the "constness" of the function arguments
- The vectors and matrices should not be changed → should be passed as `const` references

E8:T4: "Trapezoid Printing"

- Read the tasks carefully :)
- Careful with `print_diamond` and `print_hourglass`: Special cases are required if the widths are 0, otherwise there is infinite output

Questions?

3. Objectives

Objectives

- ☐ be able to understand the differences between pointers and references
- ☐ be able to trace and write programs with pointers
- ☐ be able to write programs that use dynamic memory
- ☐ be able to implement simple containers

4. & vs *

The meanings of &

The symbol & has many meanings in C++ which is confusing
It has *3 different meanings* depending on its position in code:

The meaning of &

1. as AND-operator
   ```
   bool z = x && y;
   ```
2. to *declare* a variable as an alias
   ```
   int& y = x;
   ```
3. to get the *address* of a variable (address-operator)
   ```
   int *ptr_a = &a;
   ```

The meanings of ∗

Ditto with the symbol ∗.

The meaning of ∗

1. as (arithmetic) multiplication-operator
   ```
   z = x * y;
   ```
2. to declare a pointer variable
   ```
   int* ptr_a = &a;
   ```
3. to access a variable via its pointer (dereference-operator)
   ```
   int a = *ptr_a;
   ```

Questions?

5. References vs Pointers

References

```
void references(){
  int  a = 1;
  int  b = 2;
  int& x = a;
  int& y = x;
  y = b;

  std::cout
  << a << " "
  << b << " "
  << x << " "
  << y << std::endl;
}
```

Trace program and write expected output, if the function is called

2 2 2 2

Pointers

```
void pointers(){
  int  a = 1;
  int  b = 2;
  int* x = &a;
  int* y = x;


  std::cout
  << a << " "
  << b << " "
  << x << " "
  << y << std::endl;
}
```

Trace program and write expected output, if the function is called

```
1 2 0x7ffe4d1fb904 0x7ffe4d1fb904
```

(The addresses could be different each time when called!)

Pointers und Adressen

```
void ptrs_and_addresses(){
  int  a = 5;
  int  b = 7;

  int* x = nullptr;
  x = &a;

  std::cout << a << "\n";
  std::cout << *x << "\n";

  std::cout << x << "\n";
  std::cout << &a << "\n";
}
```

Trace program and write expected output, if the function is called

```
5
5
0x7ffe4d1fb914
0x7ffe4d1fb914
```

(The addresses could be different each time when called!)

Questions?

6. this->

What the f*&k is `this->`?

The meaning of `this->`

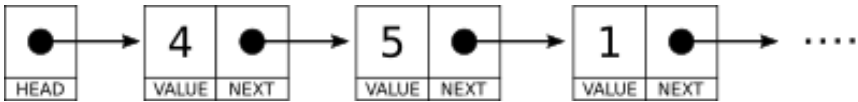
`this->` has two parts

- **`this`**
  - is a pointer to the current object (class or struct)
  - so it is of type **`T*`**
- **`->`**
  - is a cool looking operator
  - **`this->member_element`** is equivalent to **`*(this).member_element`**
  - the arrow operator dereferences a pointer to an object in order to access one of its members (functions or variables)

7. Dynamic Data Structures & Iterators

"Our-List" Primer I

We will implement (parts of) our own linked-list



- A list is comprised of "blocks" of **`lnodes`** with one **`lnode`** always pointing to the next
- But what even is an **`lnode`**?
- Answer: A struct made up of an **`int value`** and an **`lnode`**-pointer

"Our-List" Primer I

First task: Implement a constructor that initializes a new list with iterators

- We want to be able to write `our_list my_list(begin, end);`
- Idea: Use the iterators to add new `lnodes` to the list
- How can we access the different elements?
  - Access to Value of the `lnode` that the iterator is pointing to:

    `*it`
  - Next `lnode` in line:

    `node->next`
  - Create a pointer to a new `lnode`:

    `new lnode{value, pointer}`

    Remember: `new T` returns a `T*`

Exercise "`our_list::init`"

- Open "`our_list::init`" on **code** expert
- Think about how you would approach the problem with pen and paper
- Implement a solution (optionally in groups)

Exercise "`our_list::init`" (Solution)

```
our_list::our_list(our_list::const_iterator begin,
                   our_list::const_iterator end) {
  this->head = nullptr;                      // Init head (safely)

  if (begin == end) {return;}                // Case: empty list

  our_list::const_iterator it = begin;       // Adding first element
  this->head = new lnode { *it, nullptr };
  ++it;
  lnode *node = this->head;

  for (; it != end; ++it) {                  // Adding remainig elements
    node->next = new lnode { *it, nullptr };
    node = node->next;
  }
}
```

Questions?

"Our-List" Primer II

Second task: Implement a method of the class "`our_list`" that swaps a node with the next one

- You can use a similar approach to other swap functions (i.e. with a temporary variable `tmp`)
- However:
  - Use Pointers
  - What happens in the case of 0 (when the head pointer should be swapped)?
  - How can you avoid suddenly accessing memory that is not yours?

Exercise "`our_list::swap`"

- Open "`our_list::swap`" on **code** expert
- Think about how you would approach the problem with pen and paper
- Implement a solution (optionally in groups)

Exercise "`our_list::swap`" (Solution)

```
void our_list::swap(unsigned int index) {

  if (index == 0) {

    assert(this->head != nullptr);
    assert(this->head->next != nullptr);

    lnode* tmp = this->head->next;
    this->head->next = this->head->next->next;
    tmp->next = this->head;
    this->head = tmp;

  }
```

Exercise "`our_list::swap`" (Solution)

```
else {  lnode* prev = nullptr;
        lnode* curr = this->head;

        while (index > 0) {                  // Find the element
          prev = curr;
          curr = curr->next;
          --index;
        }

        assert(curr != nullptr);
        assert(curr->next != nullptr);

        lnode* tmp = curr->next;             // Swap with the next one
        curr->next = curr->next->next;
        tmp->next = curr;
        prev->next = tmp;                   }}// two '}' to close function
```

Questions?

8. Outro

General Questions?

Till next time!

Cheers!