Session 11
Dynamic
Data Structures I

Informatik I

Lan Zhang

15.05.2025

Schedule

1. Exercise Feedback
2. Theory Recap
   - Memory Management
   - Shared & Unique Pointers
3. In-Class Code Examples
   - Box (copy)
   - Shared Pointers

Dynamic Memory Allocation

for data structures, where elements can be added / deleted during program runtime

```
T* p = new T
```

- Memory for a new object of type T is allocated
- p is a pointer to the address of the object

```
T* p = new T[n]
```

- Memory for a new array with elements of type T is allocated
- p is a pointer to the first element of the associated memory range

Static array: Size is determined at compile time and cannot be changed.
Memory efficient. (e.g. C-array, std::array<T,N>)

Dynamic array: Size is determined at runtime. (e.g. std::vector<T>)

Dynamic Memory Allocation

Objects that are created with `new`

- are stored in the heap, local variables are stored in the stack
- exist until they are explicitly deleted with `delete`
- must be accessed via a pointer
- can be accessed globally

```
delete p;
```

- p points to an object previously created with `new`

```
delete[] p;
```

- p points to an array previously created with `new`

➔ Frees the memory at address stored in p

➔ Set p to `nullptr` separately to avoid invalid memory access

Memory Management

The Rule of three: If a class manages dynamic resources, it should explicitly define the following three member functions to ensure proper copying and deletion.

Destructor	deletes elements
Copy constructor	called at initialization; makes a member-wise copy of the object by default
	`Type(const Type& t); // declaration`
Assignment operator	called after initialization; replaced object is deleted
	`Type& operator=(const Type& t); // declaration`

```
Type t0; // init by default constructor
Type* t1 = new Type(t0); Type t2(t1); Type t3 = t2; // init by copy constructor
t3 = t0; // assign by assignment operator
```

Memory Management

The Destructor

- Declaration: ~Type() { ... }
- Called automatically when `delete` is called or when scope of an object ends
- Generated automatically when none is defined

```
llvec::~llvec(){
        while (head != nullptr){
                llnode* t = head;
                head = t->next;
                delete t;
        }
}
```

Memory Management

The Copy Constructor

```
llvec::llvec(const llvec& copy){
        if (copy.head == nullptr) return;
        head = new llnode(copy.head->value, nullptr);
        llnode* prev = head;
        for (llnode* n = copy.head->next; n != nullptr; n = n->next){
                llnode* t = new llnode(n->value, nullptr);
                prev->next = t;
                prev = prev->next;
        }
}
```

*copy is the object to be copied

Memory Management

The Assignment Operator

```
llvec& llvec::operator=(const llvec& copy){
	if (this != &copy){ // no self assignment
		llvec t = copy; // copy constructor
		std::swap(head, t.head);
	} // t is deleted
	return *this;
}
```

*copy is the object to be assigned

Memory Management

Problems related to incorrect use of dynamic memory:

Dangling pointer

- Points to an invalid memory location (object gone out of scope or deleted)
- Accessing it leads to undefined behavior

Memory leak

- Value to which a pointer points to is not deallocated and thus leaked
- Causes the program to consume more memory over time
- Every `new` call should have a matching `delete`

Double-free

- `delete` is called twice on the same memory allocation
- Leads to segmentation faults

Pointer Arithmetic

Sequential Iteration:

```
for (char* it = p; it != p + size; ++it){
      \\ size = number of elements in the array
      statement;
}
```

Random Access: `p[i]== *(p + i)`

Shared Pointers

Smart pointers are wrappers around regular pointers that help prevent memory leaks by automatically managing memory.

`std::shared_ptr<T>`

- allows multiple pointers to share ownership of the same resource
- stores the number of pointers pointing to this object
- object is deleted once the count reaches 0

`std::unique_ptr<T>`

- used for exclusive ownership of an object
- non-copyable
- associated memory is automatically deallocated when it goes out of scope

In-Class Code Example

```
#include <memory>
std::shared_ptr<T> s = std::make_shared<T>(/* constructor arguments */);
s.use_count() // Check number of references
s = nullptr; // Object is deleted when ref_cnt = 0


std::unique_ptr<T> p = std::make_unique<T>();
std::unique_ptr<T> b = std::move(a); // Ownership transferred
```