



RC5

Tianle Xue

Global College - Shanghai Jiaotong University

10/31/2025



# Table of Contents

Dynamic Memory Allocation

Deep Copy

Dynamic Resizing

Linked List



## Local variable and global variable

We have seen two types of variables so far:

- ▶ Global Variables
- ▶ Local Variables

Compiler know the the space for the variable. But for dynamic memory, compiler don't know the space in advance, which means that when you create them, you need to delete them(compiler doesn't help you to do so).



## Dynamic Memory Allocation

- ▶ **new:** Reserve space for an object, initialize the object, and return a pointer to it.
- ▶ **delete:** Given a pointer to an object created by new, destroy the object and release the space previously occupied by that object.



## Example

```
1  int *a = new int; //create an int;
2  int *b = new int(5); //create an int of value 5
3  int *arr = new int[n]; //create an array of int with size of n
4  int *init_arr = new int[n]{1,2}; //create an array of int with size
    n, with the first two elements initialized
5  delete a;
6  delete b;
7  delete [] arr;
8  delete [] init_arr; //release the memory
```



# Practice

Any problem with the following code?

```
1  int* p1 = new int(5);  
2  int* p2 = new int(5);  
3  int* p3 = p1;  
4  p1 = p2;  
5  delete p1;  
6  delete p2;  
7  delete p3;
```



## Answer

1. The memory for p2 has been deleted for twice.



## Possible reasons for memory leak

- ▶ delete an object that is not created by `new`.
- ▶ delete an object more than once.
- ▶ Didn't delete an object created by `new` before the program exits.
- ▶ An object is used after deleted.





## Check Memory Leak

1. Use: `valgrind--leak-check=full ./program <args>`
2. Install: `sudo apt-get install valgrind`



# Function Overloading

- ▶ **Static polymorphism:** function overloading (determined at compile time, in the same class).
- ▶ **Dynamic polymorphism:** virtual mechanism of inheritance (determined at runtime, in different classes).

Function overloading: Two different functions with exactly the same name, but different argument count or argument types.



## Rules

- ▶ Default arguments should be put at the end of the argument list.
- ▶ The return type can be different as long as the argument lists are different.
- ▶ The overloaded functions must be distinguishable when being called.

```
1 // 1. Basic function
2 void foo(int x);
3 // 2. Different argument types
4 void foo(double x);
5 // 3. Different argument counts
6 void foo(int x, double y);
7 // 4. With default value
8 void foo(int x, int y, int z = 0.0);
```



## Practice

Which of the following functions are correctly overloading void foo(int x)?

```
1 // 5.  
2 int foo(int x);  
3 // 6.  
4 void foo(int x = 0, double y);  
5 // 7.  
6 void foo(int x, char y = 'z');  
7 // 8.  
8 void Foo(int x);
```



## Overloaded Constructor

```
1 // player.h
2 class Player {
3     int num;
4     Card* cards;
5 public:
6     Player();           // default constructor
7     Player(int init_num = 5); // overloaded constructor
8 };
```

```
1 // player.cpp
2 Player::Player() : num(5), cards(new Card[num]) { ... }
3
4 Player::Player(int init_num)
5     : num(init_num), cards(new Card[init_num]) {
6     // do not write the default value in implementation
7 }
```



## Destructor

When we have used `new` to create some members of the class, we should delete them in the destructor if they haven't been deleted in the previous part of the program.

- ▶ The name of the destructor is “~ + class name”, e.g. `~Player()`.
- ▶ The destructors for any ADTs declared locally within a block of code are called automatically when the block ends.

```
1 Player::~~Player() {  
2     delete [] cards;  
3 }
```



# Table of Contents

Dynamic Memory Allocation

Deep Copy

Dynamic Resizing

Linked List



## Shallow Copy Example (box.h)

```
1  #ifndef BOX_H
2  #define BOX_H
3
4  #include <iostream>
5
6  class Box {
7  private:
8      int* length;
9      int* width;
10     int* height;
11
12 public:
13     Box(int l = 10, int w = 20, int h = 30)
14         : length(new int(l)),
15           width(new int(w)),
16           height(new int(h)) {}
```





## Shallow Copy Example (box.h)

```
1  ~Box() {  
2      delete length;  
3      delete width;  
4      delete height;  
5  }  
6  void set(int l, int w, int h) {  
7      *length = l;  
8      *width = w;  
9      *height = h;  
10 }  
11 #endif // BOX_H
```



## Shallow Copy Example (box.h)

```
1
2 void show() {
3     std::cout << "length_address_=" << length << std::endl;
4     std::cout << "width_address_=" << width << std::endl;
5     std::cout << "height_address_=" << height << std::endl;
6     std::cout << "Length_=" << *length << std::endl;
7     std::cout << "Width_=" << *width << std::endl;
8     std::cout << "Height_=" << *height << std::endl;
9 }
10 };
11
12 #endif // BOX_H
```



## Shallow Copy Example (main.cpp)

```
1  #include "box.h"
2
3  int main() {
4      Box box1;
5      box1.set(10, 20, 30);
6
7      Box box2 = box1;    // uses the default copy constructor
8
9      std::cout << "show_box1:" << std::endl;
10     box1.show();
11     std::cout << "show_box2:" << std::endl;
12     box2.show();
13
14     return 0;
15 }
```



## Shallow Copy Example

When we run the program, we see (typical output):

```
1 show box1:
2 length address = 0x13a60b630
3 width address  = 0x13a60b6a0
4 height address = 0x13a60b6a0
5 Length = 10
6 Width  = 20
7 Height = 30
8
9 show box2:
10 length address = 0x13a60b630
11 width address  = 0x13a60b6a0
12 height address = 0x13a60b6a0
13 Length = 10
14 Width  = 20
15 Height = 30
```



## Rule of the Big Three

Specifically, if you have any dynamically allocated storage in a class, you *must* provide:

- ▶ A destructor
- ▶ A copy constructor
- ▶ An assignment operator



## Basic Idea: copyFrom

- ▶ We don't want box1 and box2 to “share” the length, width and height that are stored in the same block of memory.
- ▶ Instead, we want them to have their own length, width and height stored in different blocks of memory.

Two steps:

### 1. Allocate new memory

```
1 length = new int;  
2 width  = new int;  
3 height = new int;
```

### 2. Put the values in

```
1 *length = *box.length;  
2 *width  = *box.width;  
3 *height = *box.height;
```



## Basic Idea: copyFrom

Put them together: copyFrom

```
1 void copyFrom(const Box& box) {  
2     length = new int;  
3     width  = new int;  
4     height = new int;  
5  
6     *length = *box.length;  
7     *width  = *box.width;  
8     *height = *box.height;  
9 }
```



## Copy Constructor

Simply call copyFrom:

```
1 Box(const Box& box) {  
2     copyFrom(box);  
3 }
```

Question: can we implement copy constructor as follows?

```
1 Box(const Box& box)  
2     : length(new int), width(new int), height(new int) {  
3     copyFrom(box);  
4 }
```





## Assignment operator

```
1 Box& operator=(const Box& box) {  
2     if (this != &box) {  
3         delete length;  
4         delete width;  
5         delete height;  
6         copyFrom(box);  
7     }  
8     return *this;  
9 }
```

Think: why not just

```
1 copyFrom(box);
```



## Copy Constructor vs Assignment Operator

```
1 // main.cpp
2 #include "box.h"
3
4 int main() {
5     Box box1;
6     box1.set(10, 20, 30);
7     Box box2 = box1;
8     Box box3;
9     box3 = box1;
10
11     std::cout << "show_box1:" << std::endl;
12     box1.show();
13     std::cout << "show_box2:" << std::endl;
14     box2.show();
15     return 0;
16 }
```



## Some basic difference

Copy constructor	Assignment operator
<p>It is called when a new object is created from an existing object, as a copy of the existing object.</p> <p>It creates a separate memory block for the new object.</p> <p>C++ compiler implicitly provides a copy constructor if no copy constructor is defined in the class.</p>	<p>This operator is called when an already initialized object is assigned a new value from another existing object.</p> <p>It does not create a separate memory block or new memory space.</p> <p>A bitwise copy gets created if the assignment operator is not overloaded.</p>



# Table of Contents

Dynamic Memory Allocation

Deep Copy

Dynamic Resizing

Linked List



## Motivation

1. In many cases, we do not know the length of an array in advance. We might resize it when using it.
2. So, our list should "grow" automatically, namely dynamic resizing.



## Steps to implement "grow"

- ▶ Allocate a new array with a larger size.
- ▶ Copy the elements from the old array to the new array.
- ▶ Delete the old array.
- ▶ Modify `num__elements` and `capacity`, depending on the invariant.



## Example

```
1 void grow() {  
2     int* new_array = new int[capacity * 2];  
3     for (int i=0; i<num_elements; i++){  
4         new_array[i] = array[i];  
5     }  
6     delete array;  
7     array = new_array;  
8     capacity *= 2;  
9 }
```



# Table of Contents

Dynamic Memory Allocation

Deep Copy

Dynamic Resizing

Linked List





# Implementation

```
1 struct Node{  
2     int data;  
3     Node* next;  
4 };
```



## Basic operation of linked list (Single-ended singly linked list)

1. Insertion (beginning/end), deletion, query...
2. Recommendation: Draw some graphs by hand and then implement it step by step.



## Insertion at beginning

```
1 void push(Node* first , int new_data){  
2     Node* new_node=new Node;           //step 1: allocate new to-be-  
        inserted node  
3     new_node->data=new_data;           //step 2: put in the data  
4     new_node->next=first;              //step 3: Make next of new node as  
        the old head  
5     first=new_node;                   //step 4: Move the head to point to  
        the new_node  
6     //Attention: step 4 cannot move before step 3!  
7 }
```



## Insertion at end

```
1 void append(Node* head, int new_data){
2     //1. allocate a new node and put in the data
3     Node* new_node= new Node;
4     new_node->next=nullptr;    // for the last node, the next of it
        should be NULL
5     new_node->data=new_data;
6     // IMPORTANT: check the boundary case!
7     if (!head){
8         head=new_node;
9         return;
10    }
11    Node* last=head;
12    while (last->next){
13        last=last->next;    //traverse
14    }
15    last->next=new_node;
16    return;
```



## Delete from the beginning

```
1 void remove(Node* head) {  
2     Node* victim= head;  
3     head=head->next;  
4     delete victim; //why don't we delete head directly?  
5 }
```



## Double-ended singly linked list

```
1 class IntList{  
2     Node* first;  
3     Node* last;  
4 public:  
5     ...  
6 };
```

1. last points to the last node of the list if it's not empty, otherwise nullptr.



## Insert an element at the end of list efficiently

```
1 void insert(int new_data){  
2     Node* new_node = new Node;  
3     new_node->data=new_data;  
4     new_node->next=nullptr;  
5     if(!first){  
6         first=last=new_node;  
7     } else {  
8         last->next=new_node;  
9         last=new_node;  
10    }  
11 }
```



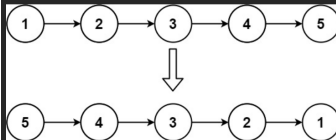
# Interesting Exercise

## 206. Reverse Linked List

Easy Topics Companies

Given the `head` of a singly linked list, reverse the list, and return *the reversed list*.

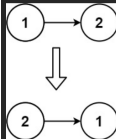
Example 1:



**Input:** `head = [1,2,3,4,5]`

**Output:** `[5,4,3,2,1]`

Example 2:







# Interesting Exercise

## 234. Palindrome Linked List

Easy

Topics

Companies

Given the `head` of a singly linked list, return `true` if it is a *palindrome* or `false` otherwise.

### Example 1:

**Input:** `head = [1,2,2,1]`

**Output:** `true`

### Example 2:

**Input:** `head = [1,2]`

**Output:** `false`

### Constraints:

- The number of nodes in the list is in the range `[1, 105]`.
- `0 <= Node.val <= 9`



# Interesting Exercise

## 21. Merge Two Sorted Lists

Solve

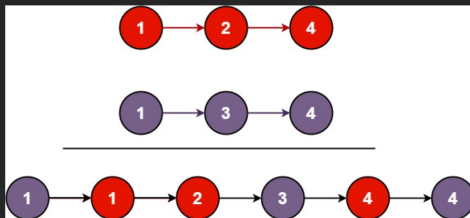
Easy Topics Companies

You are given the heads of two sorted linked lists `list1` and `list2`.

Merge the two lists into one **sorted** list. The list should be made by splicing together the nodes of the first two lists.

Return the head of the merged linked list.

Example 1:



Input: list1 = [1,2,4], list2 = [1,3,4]

Output: [1,1,2,3,4,4]

Example 2:

Input: list1 = [], list2 = []

Output: []



## References

- [1] ECE2800J-25SP Wu Haoyang RC8
- [2] ECE2800J-25SP Wu Haoyang RC9
- [3] ECE2800J-25SP Hu Chengrui RC8
- [4] ECE2800J-25SP Hu Chengrui RC9
- [5] ECE2800J-25FA Lecture Slides