RC5

Tianle Xue

Global College - Shanghai Jiaotong University

12/10/2025

Table of Contents

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

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

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

Practice

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

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

Overloaded Constructor

```cpp
// player.h
class Player {
    int num;
    Card* cards;
public:
    Player();                     // default constructor
    Player(int init_num = 5); // overloaded constructor
};
```

```cpp
// player.cpp
Player::Player() : num(5), cards(new Card[num]) { ... }

Player::Player(int init_num)
    : num(init_num), cards(new Card[init_num]) {
    // do not write the default value in implementation
}
```

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 "$\sim +$ class name", e.g. `~Player()`.
- The destructors for any ADTs declared locally within a block of code are called automatically when the block ends.

```cpp
Player::~Player() {
    delete[] cards;
}
```

The constructor ensures that the object is a legal instance of its class and the destructor's job is to destroy the object.

Table of Contents

Example

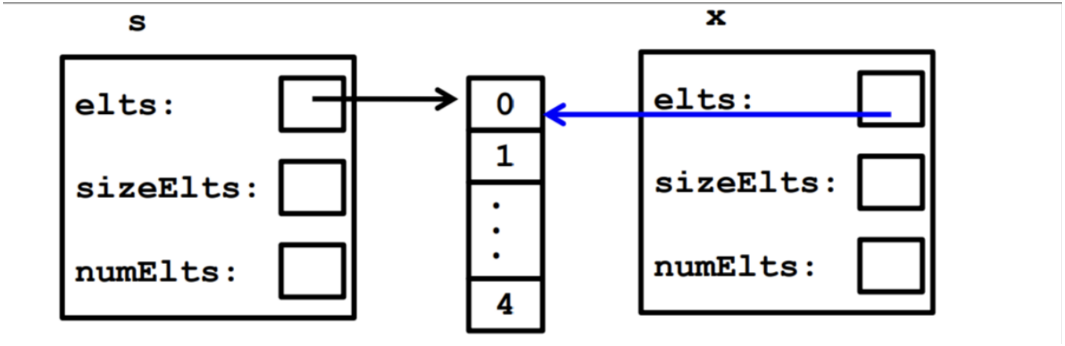
See the given example of the main.cpp and box.h .
When we run the program main.cpp, we see (typical output):

```
show box1:
length address = 0x13a60b630
width address  = 0x13a60b6a0
height address = 0x13a60b6a0
Length = 10
Width  = 20
Height = 30

show box2:
length address = 0x13a60b630
width address  = 0x13a60b6a0
height address = 0x13a60b6a0
Length = 10
Width  = 20
```

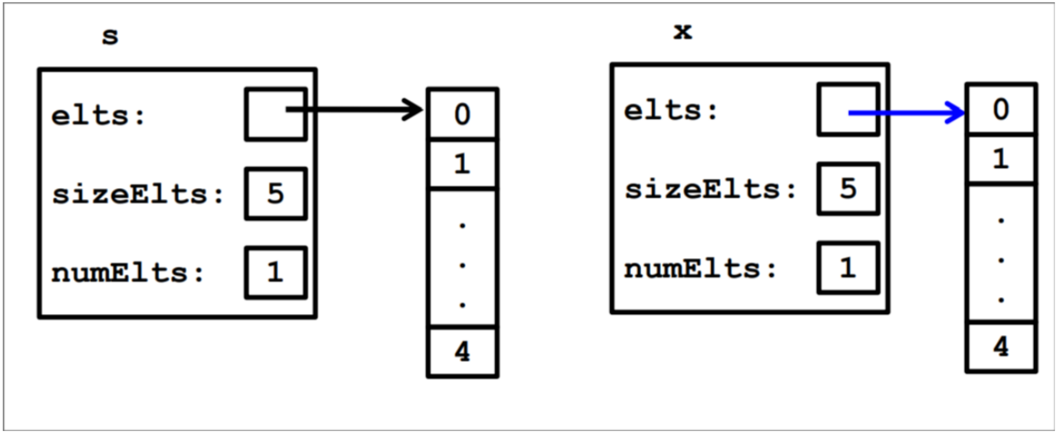
Shallow Copy

Shallow copy: Copy the values of the member variables from one object to another.



Deep Copy

Deep copy: Copy while allocating new memory for the dynamic member variables.



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

```
length = new int;
width  = new int;
height = new int;
```

2. Put the values in

```
*length = *box.length;
*width  = *box.width;
*height = *box.height;
```

Basic Idea: copyFrom

Put them together: copyFrom

```
void copyFrom(const Box& box) {
    length = new int;
    width  = new int;
    height = new int;

    *length = *box.length;
    *width  = *box.width;
    *height = *box.height;
}
```

Copy Constructor

Simply call copyFrom:

```
Box(const Box& box) {
    copyFrom(box);
}
```

Question: can we implement copy constructor as follows?

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

Assignment operator

```
Box& operator=(const Box& box) {
    if (this != &box) {
        delete length;
        delete width;
        delete height;
        copyFrom(box);
    }
    return *this;
}
```

Think: why not just

```
copyFrom(box);
```

Copy Constructor vs Assignment Operator

```
// main.cpp
#include "box.h"

int main() {
    Box box1;
    box1.set(10, 20, 30);
    Box box2 = box1;
    Box box3;
    box3 = box1;

    std::cout << "show␣box1:" << std::endl;
    box1.show();
    std::cout << "show␣box2:" << std::endl;
    box2.show();
    return 0;
}
```

Some basic difference

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

Table of Contents

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

```
void grow(){
  int* new_array = new int[capacity*2];
  for(int i=0;i<num_elements;i++){
    new_array[i]=array[i];
  }
  delete array;
  array=new_array;
  capacity*=2;
}
```

Interesting problem

What about resizing the array when many elements are deleted?

Table of Contents

Implementation

```
struct Node{
  int data;
  Node* next;
};
```

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

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

Insertion at beginning

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

Insertion at end

```cpp
void append(Node* head, int new_data){
  //1. allocate a new node and put in the data
  Node* new_node= new Node;
  new_node->next=nullptr;   // for the last node, the next of it
      should be NULL
  new_node->data=new_data;
  // IMPORTANT: check the boundary case!
  if(!head){
    head=new_node;
    return;
  }
  Node* last=head;
  while(last->next){
    last=last->next;   //traverse
  }
  last->next=new_node;
  return;
```

Delete from the beginning

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

Double-ended singly linked list

```
class IntList{
  Node* first;
  Node* last;
public:
  ...
};
```

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

```
void insert(int new_data){
  Node* new_node = new Node;
  new_node->data=new_data;
  new_node->next=nullptr;
  if(!first){
    first=last=new_node;
  } else {
    last->next=new_node;
    last=new_node;
  }
}
```

Interesting exercise

See the attached file.

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

[6] ECE2800J-25SP Wu Haoyang Final RC