# COMP 322 Lecture 5 - Memory Management

Junji Duan

2024/2/2

---

## Today's Outline

- Automatic memory
- Dynamic memory
- Cleaning the mess
- Arrays vs pointers

---

## Automatic memory

- Amount of memory to be allocated should be known in advance at compile time
- Any "normal" variable declaration uses automatic memory allocation (in some textbooks they refer to it as static memory. Do not confuse it with the static keyword. In this context they mean automatic memory)
  - int var; compile time vs run time
  - int myArray[10];
- Automatic memory is automatically liberated when the variable goes out of scope
  - No need for programmer's intervention
- Automatic memory is being allocated from the "*Stack*" (fancy name for a region in memory)

## Automatic memory - limitations

```
// main function
int main()
{
    int x = 5;
     // some code affecting x
    if (x <= 22)
    {
        int y = 12;
        // do something ...
    }
    cout << y; // ??
}
```

This code won't compile ...

- Automatic variables cannot survive beyond their scope
- To be able to use the variable y, it should be declared global
  - What if y's creation is decided at runtime and not during compile time?

```
// main function
int main()
{
    int *x;
    int a = 12;
    if (a <= 22)
    {
        int y = 13;
        x = &y;
    }
    cout << *x; // ??
}
```

- Using pointers to gain access to y's memory location will lead to undefined behavior
- This code does compile and run, however, there is no guarantee for what the value of *x would be

- What if the size of the memory needed isn't known until runtime?
  - Example: size of an array depends on user's input
- What if we needed a function or a block of code to return a pointer to a valid memory that was being declared within the function or block (similar to previous example)
- What if we have limited memory due to system constraints?
  - We cannot book in advance too much memory and keep it reserved for all the lifetime of the program

## Dynamic memory

- Dynamic memory allocation does not require the amount of memory needed to be known before run time
- Dynamic memory is being allocated from the "*Heap*" (fancy name for a region in memory also known as the free store)
- Memory management is the responsibility of the programmer
  - Great power requires great wisdom
  - No Java style garbage collection

- Use **new** operator to dynamically allocate memory malloc memory allocation
  - Similar to malloc in C language
- Use **delete** operator to free the allocated memory when it is not needed anymore
  - Similar to free in C language
- Use **new[ ]** operator to dynamically allocate a sequence of memory locations
  - Example: to allocate an array of elements
- Use **delete [ ]** operator to dynamically free a sequence of memory locations
- **delete** only memory allocated with **new**
- **delete [ ]** only memory allocated with **new [ ]**

C++ Style memory allocation — same — C Style memory allocation

```
// main function
int main()
{
    int *x = new int;
    *x = 5;
    // do something with x
    cout << *x;
    delete x;
}
```

```
// main function
int main()
{
    int *x = (int*)malloc(sizeof(int));
    *x = 5;
    //do something with x
    cout<<*x;
    free(x);
}
```

- **Note that, unlike malloc, operator new is type-aware so no need for explicit type casting**

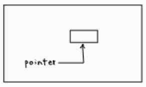
What's wrong with this code? (the memory was never freed which will lead to memory leak)

内存从未被释放，导致内存泄露

```
#include <iostream>
using namespace std;

int* getIntPointer()
{
    int *ptr = new int;
    return ptr;
}

// main function
int main()
{
    int *x = getIntPointer();
    *x = 5;
    cout << *x;
}
```




need delete

# Dynamic memory - pointer / array notation

```
#include <iostream>
using namespace std;

int* getIntArray(int size)
{
    int *ptr = new int[size];
    return ptr;
}

// main function
int main()
{
    int *x = getIntArray(5);

    for(int i=0; i<5; i++)
    {
        *(x+i) = i;
    }

    for(int i=0; i<5; i++)
    {
        cout<<*(x+i)<<endl;
    }

    delete [] x;
}
```

new int; 1 element

new int[10]; 10 elements

# Relationship between arrays and pointers

- Arrays can be compared to constant pointers

```
// main function
int main()
{
    int x = 5;
    int *var = &x;
    cout << var << endl;

```

Output is:

- 0x7ffee9c5c934

| 3 | 5 | 7 |
|---|---|---|

```
// main function
int main()
{
    int x[3] = {3, 5, 7};
    cout << x << endl;
    cout << &x[0] << endl;
    cout << &x[1] << endl;
}
```

address of array is the address of first element of the array.

Output is:

- 0x7ffe2caee980
- 0x7ffe2caee980
- 0x7ffe2caee984

- Arrays can be compared to constant pointers

```
// main function
int main()
{
    int array[3] = {3, 5, 7};
    int *ptr;
    ptr = array; // similar to ptr = &array[0]
    cout << ptr[1] <<endl;
}
```

- Output: 5

- Arrays can be compared to constant pointers

```
// main function
int main()
{
    int array[3] = {3, 5, 7};
    int *ptr;
    ptr = array; // similar to ptr = &array[0]
    cout << ptr[1] <<endl;

    int array2[3] = {3, 5, 7};
    array2 = ptr; // this will not compile
}
```

array is constant pointer, so array can't point to something else.

# Comparison between arrays and pointers

- Size of array should be predetermined in advance and cannot be resized after declaration
- The memory location of an array is fixed and cannot be changed after its declaration
- C++ supports other types of more robust arrays that we will discuss later (vector, map, etc.)

- Pointers can point to a chunk of memory of any size and this can also change during execution
- Unless declared constant, pointers can point to a different memory location later on during execution

# Dynamic memory - initialization

- Operator new can initialize the newly created objects
  - int *i = new int (2); // creates one element and initializes it to 2
  - Same as:
    - int *i = new int;
    - *i = 2;

    [ ] for number of element
  - Do not confuse with:
    - int *i = new int [2]; // creates an array of 2 uninitialized elements
    - int *i = new int [2](); // creates an array of 2 elements initialized to 0
    - int *i = new int [2]{3,5}; // creates an array of 2 elements initialized to 3 and 5. This is valid since C++11.

# Common mistakes

- "Writing in C++ is like running a chainsaw with all the safety guards removed" — by Bob Gray; cited in *Byte* (1998) Vol 23, Nr 1-4. p. 70
- "In C++ it's harder to shoot yourself in the foot, but when you do, you blow off your whole leg" — by Bjarne Stroustrup the creator of C++


- Using delete to free memory allocated by new [ ] → memory leak
- Forgetting to free memory → memory leak
- Dereferencing a pointer after deleting it → from undefined behavior to crash
- Deleting the same pointer more than once → This will cause undefined behavior and weird crashes from the 5th dimension :)

delete x

```
// main function
int main()
{
    int *x = new int;
    int *y = x;
    *y = 7;
    cout<<*x;
    delete x;
    delete y;
}
```

# C++11 Smart Pointers

- Note that since C++11 the standard library added more ways to manage dynamic memory in a safe way
    - auto_ptr // not deprecated but use unique_ptr instead
    - shared_ptr
    - unique_ptr // enhancement of auto_ptr
    - weak_ptr
- Wrappers to manage the lifetime of dynamically created objects and provide a garbage collection like environment
- We will get back to this after we cover Classes in C++

# Pointer to Pointer - Linked List Example

```cpp
#include <iostream>
using namespace std;

struct Element {
    int data;
    Element* next;
};


void appendElement (Element** list, Element** tail, int data){
    Element* e = new Element;
    e->data = data;
    e->next = NULL;

    if (!(*list)){ // if list is empty
        *list = e;
        *tail = e;
    }
    else{ // append to the end
        (*tail)->next = e;
        *tail = e;
    }
}

void printList (Element* list){
    while(list){
        cout << list->data << " ";
        list = list->next;
    }
}
```

```cpp
void deleteList(Element** list){
    Element* next;
    while(*list){
        next = (*list)->next;
        delete *list;
        *list = next;
    }
}

int main() {
    Element* list = NULL;
    Element* tail = NULL;
    appendElement (&list, &tail, 5);
    appendElement (&list, &tail, 10);
    appendElement (&list, &tail, 15);
    printList(list);
    deleteList(&list);
    return 0;
}
```