Practical Session #02 - C++ Memory Handling

Tom Mahler
In This Recitation We’ll Cover

- Memory Model
- Pointers
- Pointers Arithmetic
- Arrays
- C++ References
- Memory Handling
- Pointers Dangers
Memory Model
Memory Model

- Each process is running in its own memory space

- A memory space is the range of addresses that can be described in a single word
  - For example: in a 32bit computers, the memory space is $0 - (2^{32} - 1)$
Memory Model: Stack & Heap

- There are two "kinds" of memory available to a process:
  - Stack:
    - Stores local variables of the function
    - Removed once the function ends!
  - Heap:
    - Contains dynamically allocated variables
    - Stays once the function ends, unless cleared before!

Pointers

- A pointer is a **variable** that holds the address of some other variable
- In Java, you **can’t** directly access the memory of such object! Only by reference
- In C++, you **can** access them directly! Alter them any way you like, without any limitations!
  - **Benefit**: more control than Java
  - **Drawback**: memory leaks, memory corruption

- A 64-bit computer can address $2^{64}$ **bytes**!
- Each memory cell is 1 **byte** (byte-accessible machines - most of the machines today)
- Each pointer takes a static size of 4 bytes in a 32bit OS, and 8 bytes in a 64bit OS
- Pointers **themselves** are saved on the stack
Java vs. C++

Java’s References

- String s = new String("a string");
  - The value of the variable 's' is the location of the object on the heap, but we don’t have access to this value!
- You cannot have a variable that ‘points’ to a primitive type
- String s = null;
  - means "I don’t have the address of any meaningful data"

C++ Pointers

- Have access to actual memory locations
- Can point to anything!
- The null value in C++ is the number 0 (memory address 0)
The Operators * and &

- The operator * has 2 different meanings, depending on the context in which it appears:
  - In variable definition, it means “this variable is a pointer”
  - Once the variable is defined, it means “the value at the address pointed to by the variable”

- The operator & is “the address of the given variable“

- In different contexts, these operators can represent completely different operations
  - For example, * is also the multiplication operator, and & is the bitwise and operator (as well as the reference operator, which we’ll learn later)
  - In addition, C++ supports operator overloading, which allow you to redefine these and other operators for the classes you create
Examples

See code...
Pointers to pointers

```c
int *x;
int **y;
double *z;
```
A pointer is just a 32bit (or 64bit) number, so why do we need different types of pointers (a.k.a: pointer to int, pointer to char, etc.)?

**Type safety:** telling the compiler to check that we meant what we wrote is always a good idea

Integer math operations can be used with pointers +, +=, ++, -, -=, --

However, math operations on pointers works differently
Pointers Arithmetic

- If you increment a pointer, it will be increased by the size of whatever it points to!
- Incrementing pointers will basically make it point to the “next” element in memory
- For example:
  - Adding 1 to a pointer to int actually adds $1 \times \text{sizeof(int)}$ (which happens to equal 4 on my computer)
  - Adding 5 to a pointer of type float actually adds $5 \times \text{sizeof(float)}$

```c
int *ptr = a;

*(ptr+2)

*(ptr+4)
```

```
int a[5];
```
Examples

- See code...
Arrays

- Array name is basically a `const pointer` pointing at the beginning of the array
- You can use the `[]` operator with pointers!
- Example:
  - `int A[5];`
  - Creates a memory block of 5 integers on the stack (5*sizeof(int)) where A (the pointer) points at the beginning of the array -> A[0]
Arrays Usage Example

- `int A[10];`
- `int j = A[3];`

The meaning of these lines is:

- The first line declares an array of 10 integers and sets A to be a the address of the first element
- The second line puts the value 9 in the 4th place in the array
  - The location of the 4th place in the array is "the value of A plus 3*sizeof(int)"
  - Or using pointers arithmetic:A+3
  - So this line could be written as *(A+3) = 9;.
- The third line reads a value from the 4th place in the array
  - This line could be written as int j = *(A+3).
Unlike Java, C++ arrays don't have a "length" properties!

C++ arrays are just special pointers, and they don't know the size of the memory block they are pointing to

- You need to track this information yourself if need it

For this reason, C++ arrays will not give you an "array out of bound" exception if you try to access elements beyond the allocated memory

This is very dangerous: the operation will probably work, but write or read things you didn’t intend to

For example, what does the following lines do?

- int A[10];
- A[100] = 9;
- A[-2] = 500;
Examples

▶ See code...
Strings vs. char*

- char* is another way to represent strings in C++ (it actually came first - with C!)
- char* is an array of chars
- char* arrays are terminated with ‘\0’ - “null terminated strings”
  - Denotes end of string

Example:
- char *msg= “RPI”;
- Length of string?

```
strlen(msg) == 3;
```
Convert `std::strings()` to char* Strings and Back

- `c_str()` returns a const char* representation of the string

- It returns a pointer to an internal representation
  - This is GOOD: because you don't have to manage this memory yourself
  - This is also BAD: there's no guarantee you will keep pointing to a valid c-string after additional calls to methods of the string
Examples

- See code...
Every expression in C++ is either an lvalue or an rvalue:

- lvalue refers to an object that persists beyond a single expression
- rvalue refers to a temporary value that does not persist beyond the expression that uses it

A good indication should be the location of the expression

- The rvalue can not appear on the left side of an expression

The compiler might use the lvalue / rvalue definitions to help you identify issues in your code
lvalue Reference

- Basic references

- They behave like a const pointer, this means that they point to a specific location but you cannot change the location

- With references, you cannot even access that location

- This means that lvalue references must be initialized with a real value (not null)

- lvalue references allows the coder to pass objects similarly to java
Examples

- See code...
Memory Handling

- Up till now we allocated memory on the stack
- Now we learn how to allocate memory on the heap
- This is done with the *new* operator
- *new* allocates memory on the heap, initializes it and returns a pointer to it
Memory Handling

- All the memory allocated by `new` must be freed, or you will get a memory leak.

- The `delete` operator takes a pointer to an allocated space on the heap, and frees it.
  - The memory is returned to the OS.

- However, you need to be very careful: don't try to delete a pointer not allocated by `new`!

- Use safe deletes as shown in the code to make sure the pointer exists before you delete.
  - Safe delete is recommended when a pointer is shared in many functions.
Memory Handling Fail

- Memory allocation may fail

- There might not be enough space on the heap, in which case `new` will throw the `std::bad_alloc` exception

- You should only catch it if the program is able to recover from such a state

- We will learn about catching exception later
Examples

See code...
Arrays on the Heap

- We've seen before that arrays are just pointers to a block of memory.
- You can use the new[] operator to allocate a block of memory on the heap.
- Use delete[] to free the allocated memory.
  - Note that delete[] does not call delete on each object in your array!
  - This means that if you have an array of objects, you must delete all objects one by one and delete[] the array at the end.
```c
int *A = new int[10];
// allocate 10 integers on the heap
int *B = A + 1;
A[0] = 1;
A[1] = 2;
B[1] = 3;
delete[] A;
// free the allocated memory
```
Examples

- See code...
Dangling Pointer

- A pointer that points to nothing
- Example:
  - int *p, *q;
  - p = new int;
  - q = p;
  - delete q;
  - *p = 3; //illegal assignment!

- p and q point to the same location, q is deleted, results with p becoming a dangling pointer!
Memory Leak

- Memory leak is when you remove the reference to the memory block, before deleting the block itself.

- Example:
  - int *p = new int;
  - p = NULL; // or a new other value

- p points at memory location of type int
- p changed to point at null
- Result? **Memory leak!**

- Must free memory block before changing reference
Memory Leak

- Memory leak occurs when the program fails to release memory that is no longer needed.
- A memory leak can diminish the performance of the computer by reducing the amount of available memory.
- Eventually, in the worst case, too much of the available memory may become allocated and all or part of the system or devices stops working correctly, the application fails, or the system slows down.

Use `valgrind` with `-leak-check=yes` option if your implementation has memory leaks.

- [Valgrind User Manual](#)
- [The Valgrind Quick Start Guide](#)
- [Graphical User Interfaces](#)

* This is why the `-g` flag is used when debugging!
Examples

See code...
Pointers Dangers Summary

- Uninitialized pointers
- Dereferencing a null pointer
- Dereferencing a deleted pointer
- Dereferencing a dangling pointer

Use delete for freeing memory allocated with new and delete[] for memory allocated with new[]