Pointers, references

- Pointers give a program access to the actual (virtual) memory address of items, whereas references provide a mechanism for indirect access without explicitly providing a memory address
- The layer of indirection in references allows most HLL to implement information handling in a black box fashion hidden from developers, safeguarding against accidental or deliberate misuse
- C/C++ relatively unusual (for HLL) in giving direct access to (virtual) memory through addresses
- Pointers are powerful tools, but with power comes risk

Uses of pointers/references

- Allows use of dynamic (heap) allocated resources, whether implicitly (e.g. objects in most OO languages) or explicitly (e.g. malloc, calloc, etc in C)
- Allows passing references/pointers to subroutines, providing ability for subroutine to access (and possibly alter) the item without explicit need to copy item back/forth on stack
- Allows for distinction between shallow and deep copies and ability to take advantage of whichever is most appropriate

Dynamic (heap) allocation

- Allows item lifetime to persist across function calls without need for globals
- To be effective, there needs to be means of communicating location of dynamic item (i.e. via pointer or reference) and a means of deallocating item when no longer needed
- Deallocation can be handled automatically (requires support implementation at compiler level) or explicitly by programmer (greater flexibility but creates risk of memory leaks, wild pointers, dangling pointers, null pointers, etc)
- One reason modern OS segregate programs into their own virtual memory spaces is to counteract the many failures of dynamic memory handling techniques
- we’ll look deeper into both garbage collection and dynamic memory allocation later

Typed pointers, references

- When people are first introduced to pointers it often seems strange that they need an associated type, after all, it is simply a memory address, right?
- But when we go to an address, we just see a bunch of bits – we need to know what that bit pattern represents, which depends on what kind of data we think is stored there (bit patterns stored differently for ints than floats etc)
- For languages (like C) allowing access to memory through numeric addresses, type checking vastly complicated by difficulty of knowing the data at the specified address is of an appropriate type

Pointer operations

- For languages supporting pointers, basic operations needed include ability to take address of something, copy pointers, compare pointers for equality, dereference pointers
- Might be reasonable to allow numeric comparisons of memory addresses, e.g. is X stored earlier in memory than Y?
- More complicated issue is do we allow pointer math?

If I take address of X, can I add offsets to it, effectively allowing me to explore forward/backward in memory?

Can I explicitly use a number as a pointer address, allowing me to jump anywhere?
Implications for safeguarding memory from accidental/deliberate misuse