**Memory, Linked Structures & ADTs**

* Memory model
  + Run-time stack manages storage needs for each pending function call (local variables & parameters)
  + Heap (“freestore”) holds actual storage for all dynamically allocated objects (e.g. malloc, new)
    - Must return this storage via free/delete
* Struct/class
  + Structs and classes are **equivalent** in C++
  + Instances of structs/classes are called **objects**; they can be created by:
  + Direct instanstiation
    - structName s;
    - s.field; s.method();
    - Space is allocated on the **run-time stack**, but instances disappear at the end of their defining scope – problem for being used for linked structures
  + Dynamic instantiation
    - structName \*s = new structName;
    - s->field; s->method();
    - delete s;
    - Space for the object is allocated on the **heap**; it persists until explicitly deleted. The **pointer** s is allocated on the **stack**
  + Ex:

Coord a;

Coord b = a; //object copy

print(a); //parameter is Coord

printPtr(&b); //parameter is Coord\*

Coord \*p1;

p1 = new Coord;

Coord \*p2;

p2 = p1; //ptr copy

print(\*p1); //parameter is Coord

printPtr(p2); //parameter is Coord\*

delete p1;

delete p2; //error!

//Object pointed to by p2 is already deleted

* There is no pointer arithmetic in C++
* C++11 value – nullptr – meaning it’s not pointing to anything
  + After declaring a pointer, set it to point to something or set it to nullptr right away
  + Can be treated as a bool
  + Before C++11 the NULL macro was used
* **Linked list** – each **node** has a *value* and a *next* pointer
  + A list begins with a pointer to the first node, and every node’s *next* points to the next node
  + The last element’s *next* is nullptr
* **Abstract data type (ADT)** – a mathematical structure that has well-defined and widely recognizable behaviour
  + It contains data that may only be accessed in a prescribed manner, by a set of **operations**
  + Each operation has a:
    - **Signature** – which describes the parameters & the type of returned value
    - **Pre-condition** – logic statement that specifies what is assumed to be true before the operation may be applied
    - **Post-condition** – logic statement that describes the value/effect of the operation
  + **Vector/sequence** – ordered data container that allows random access to elements, but usually only allows add/delete at the end
  + **Stack** – ordered data container with no random access; last in first out
  + **Queue –** ordered data container with no random access; first in first out
  + **Set –** unordered data container that can contain a given element at most once
    - Multi-set – can contain an element more than once
  + **Dictionary/map** – unordered data container of key pairs
    - If (a, b) and (a, c) are in M, then b == c
    - Multi-map – key pairs are not unique (the above is not true^^)
  + There may be multiple ways to implement an ADT, but its abstract specification is the same
  + Often an **interface** is created around an ADT – enforces a limited access to internal details (e.g. class definitions)
  + ADTs are data containers that can hold any type of data
  + Functional programming style to create ADTs:
    - newADT operationName (oldADT, otherParam) {}
* **Stack as a linked list**:
  + initStack() – creates and returns a new empty stack to the caller (constructor)
  + push(s, enew) – creates a new stack identical to s with enew, e1, … en
  + pop(s) – returns a new stack identical to s with e2, … en
  + peek(s) – returns the first element e1
  + isEmpty(s) – returns whether the stack is empty (n == 0?)
  + Stack as a linked list:

struct Node {

string val;

Node \*next;

}

typedef Node\* Stack

Stack initStack() {

return nullptr;

}

bool isEmpty(Stack s) {

return s == nullptr;

}

Stack push(Stack s, string val) {

Node \*newNode = new Node;

newNode->val = val;

newNode->next = s;

return newNode;

}

string peek(Stack s) {

assert (!isEmpty(s));

return s->val;

}

Stack pop (Stack s) {

assert(!isEmpty(s));

Node \*newNode = s->next;

Delete s;

return newNode;

}

void nuke (Stack s) { //destructor

while(!isEmpty(s)) {

s = pop(s);

}

}

* + Memory leak:

Stack s1 = initStack();

Stack s2 = initStack();

…

s1 = s2; //reference to s1’s memory is lost

* + The details of implementation don’t matter; a stack can be implemented differently using the same interface
* **Adapter** – an interface that provides exactly what the clients need and no more
  + Prevents the client from depending on features they shouldn’t use, and protects them from possible changes in the underlying implementation
  + E.g. stack as a vector – better to create an interface to use the vector as a stack than using the vector directly
  + **Information hiding** – separate information from implementation
    - Have clients depend on only well-designed, unlikely-to-change interfaces