Iterators – basic list example

- current – refers to most recently accessed item
  - To manipulate other parts of list (not just first, last)
  - Points to NULL if list is empty
  - Note: item just before a deleted item is the most recently accessed (to set link to NULL)
- User needs way to iterate through list items

```
void advanceCurrent(ListPointer);
```

- And a way to reset current to first item again

```
void resetCurrent(ListPointer);
```

- And best have way to ask if at end of list or not

```
int hasMoreInfo(ListPointer);
```

Basic list trade-offs

- Abstraction sacrifices efficiency
  - Function calls instead of direct node access
- User has to deal with `void *` pointers
  - Easy for insert operations – any pointer is “promoted”
  - But must cast to true pointer type on return
    ```
    printf("%s", (char *)firstInfo(list));
    ```
  - And must dereference to get to real data
    ```
    printf("%d", *(int *)currentInfo(list));
    ```
- `void *` storage also inhibits some operations
  - No way for list module to search, or sort, etc., without knowing type – one complication can fix this though

What is a recursive function?

- Ans: a function that calls itself (maybe indirectly)
- Standard first example – factorial function:

  n! = n * (n-1) * (n-2) * … * 1 (for n > 0)

  – Note *recursive* pattern:

  n! = n * (n-1)! (for n > 1, and 1! = 1)

  – Translates immediately to C:

```
int factorial(int n) {
    if (n<=1)
        return 1;
    else
        return n*factorial(n-1);
}
```

Recursive solution essentials

- Always need a base case
  - a.k.a. trivial case, or smallest case
  - A way to stop; otherwise infinite recursion
    - e.g., `if (n<=1)` in factorial method
- Recursive calls converge on base case
  - i.e., problems get smaller with each recursion
    - e.g., `factorial(n-1)`
- Solution must actually solve the problem!
  - This part is most important, and the hardest to insure

Recursive Drawing Example

- Handy for some non-numerical problems too
- Drawing tick marks on a ruler:
  - base case: draw nothing (tick too small)
  - general case: draw middle tick, then draw left and right "sub-rulers" (with smaller ticks)

```
void ruler(int left, int right, int tickHeight) {
   if (not done yet) {    /* pseudocode */
      int middle = (left + right) / 2;
      draw_tick(middle, tickHeight);
      ruler(left, middle, tickHeight / 2);
      ruler(middle, right, tickHeight / 2);
   }
}
```

Recursive list printing

- Because a list is a recursive data structure
  - Idea: print info, then call function for next node (as long as there are nodes left to print)
- Simple change prints in reverse!

```
void printReverse(NodePointer n) {
    if (n->link != NULL)
        printReverse(n->link);
    printf("%s ", n->info);
}
```

- Q: how to print opening/closing parentheses?
  - One answer: use recursive auxiliary function

Towers of Hanoi (demo)

- Some solutions are especially surprising
- Move n disks from a to c; use b to hold
  - Base case: just one disk – trivial

```
if (n==1) moveOneDisk(a→c);
```

  - General case: *assume* there is a function that can move a tower of height n-1. This function!!!

```
else {
    tower(size n-1, a→b with c holding);
    moveOneDisk(a→c);
    tower(size n-1, b→c with a holding);
}
```

- Iterative solution much more difficult in this case

Top-down programming by stepwise refinement

- Typical top-level algorithm has 3 main steps:
  1. Get data
  2. Process data
  3. Show results
  - Applies to whole program, and most functions
    - For functions, get-data step usually done by parameter list, and show-results step usually done by return
- Idea is to start with top-level, then refine steps
  - e.g., steps 2.1, 2.2, … refined step 2
    - Later refinements – step 2.2.1, 2.2.2, …
    - And so on until algorithm is complete

Functions carry out the steps

- Top down programming boils down to:
  - Write the necessary sequence of steps as function calls
  - Then write the functions
    - May involve writing deeper sequences of function calls
    - So write those additional functions
      - And so on …
- Concept is called algorithm abstraction
  - Motivation is to manage complexity
    - Don’t have to consider all the details all at once
    - Solve overall problem – then sub-problems as encountered
  - Even more powerful combined with data abstraction

Choosing data structures: key part of devising a solution

- e.g., text section 5.2, lottery ticket example
  - Top-level: (1) get 6 amounts, (2) process, (3) print amount won.
- One solution – use a Table to track repetitions –
  - Refine step 2 – make Table with 1 row for each different amount, and 2 columns with the amount and repetitions
    - Further: use array of struct{amount, reps} to represent Table
  - Refine step 3 – print 0 or highest amount that has 3+ reps
- Another solution – sort array of amounts –
  - Refine step 2 – sort the 6 amounts into descending order
  - Refine step 3 – print first amount that is repeated 3 times

Testing

- Goal is to find faults
- Faults (a.k.a. bugs) cause systems to fail
  - e.g., a system crashes – the most obvious type of fault
  - e.g., a security system that allows unauthorized entry
  - e.g., a shot-down video game plane continues on path
- Can verify the presence of bugs, not their absence
  - Testing fails if no bugs are found! (a good thing really)
- Testing and debugging are separate processes
  - Testing identifies; debugging corrects/removes faults

Testing steps

- Unit tests – insure each part is correct
  - Independently test each function in each file
- Integration tests – insure parts work together
  - Test functions working together; not whole system yet
- System tests – insure system does what it is supposed to do
- More testing to do – especially for large systems
  - Includes *functional* tests, *performance* tests, *acceptance* tests, and *installation* tests

Testing approaches

- Black box testing – best by independent tester
  - Plan good test cases, and conduct *automated* tests
- Open box testing – a separate, preliminary activity
  - "Coverage testing" is the goal
    - i.e., test every line of code at least once
  - Includes unit testing and integration testing
- Regression testing – repeat tests frequently
  - Because fixing a new bug may re-introduce old ones
  - Easy to do with automated testing framework
    - i.e., special purpose programs and data files like assignment 3

Test plans

- Test a *representative sample* of normal cases
  - Usually no way to test all possibilities
    - But don't really need to – random sample of cases okay
  - At least be sure to test all normal operations
    - e.g., insert first, delete last, show current …
- Test *boundary cases*
  - Test the extremes – includes empty cases, lone cases, last case, first case, …, any other "edge" cases
    - e.g., delete from lists with 0, 1, and 2 items
- Test *error cases* too
  - e.g., test how bad input is handled – should not crash!

Assertions

- Are conditions that should *all* be true for the program to be considered correct
  - Can be used for testing correctness with logical rules
  - Can also be tested automatically in many cases
- Most important types of assertions:
  - Function "contract" clauses
    - Pre-conditions – must be true on function entry
    - Post-conditions – must be true on function exit, *if the pre-conditions were true beforehand*
    - Both sets of conditions should be made clear to users (usually as comments in header files)
  - Loop invariants – must be true on each iteration

Executable assertions

- Can verify correctness automatically
  - e.g., pre-condition of `inverse(x)` is that x is not zero
    - Let assertion check automatically
  - Must `#include assert.h`:

```
double inverse(int x) {
    assert(x != 0); /* halts with message if x == 0 */
    return 1. / x;   /* better than crashing here */
}
```

- If pre-condition fails, it's the user's fault
  - Function doesn't know what to do, so no use for if …
    - i.e., `if (x == 0)` what now? – no good answer, really

More assertions

- Can also use assert to check post-conditions
  - Should verify key effects if a function is complex
    - In this case, errors are the fault of the implementation!
- Asserting loop invariants is useful for debugging
- Q. Why assert to check your own code?
  - Answer: catch bugs early and effectively
    - Bugs appear as soon as testing begins
    - Also know where bug occurred, and maybe where to fix it
- Note: use assert as a development tool ONLY
  - Easy to turn off all assertions for final product
    - `#define NDEBUG` before `#include <assert.h>`