Session 6 : Modules

Modularization

- Also commonly known as libraries
- Large programs can be built from many modules
- A module can be a client itself and require functions from other modules
- The module dependency graph should not have any cycles
- There must be a “**root**” that acts as a client
    - This is the program that contains the Main function and is run
- Three key advantages
    1. Reusability
        - Can be re-used by many clients
        - Might be able to buy or license third-party modules or subcontract parts of the implementation
    2. Maintainability
        - Easier to test and debug a single module using a test-suite
    3. Abstraction
        - Lets client know functionality without understanding how it’s implemented
- Module in C is made of two files
    - One containing **module declarations**
    - Other containing **module definitions**

DECLARATION: introduces an identifier
DEFINITION: gives some content to an identifier

- Also contains an identifier, so a definition always acts as a declaration
- An identifier can be **declared multiple times**, but only **defined once**

Test clients

BLACK-BOX TESTING: ensures correct functionality of an application without knowledge of internal implementations
WHITE-BOX TESTING: tests all internal functionality of a module

- May include tests for implementation-specific and module-scope function

Information hiding

- Two key advantages
    1. Security
        - Prevents clients from direct access to data stored within a module
        - Client may only interact through the given interface
    2. Flexibility
        - Allows for changing the underlying implementation without affecting the client (as long as the interface remains unchanged)

OPAQUE STRUCTURES: providing an incomplete structure declaration

- Compiler does not know how much memory to allocate for a structure
- Only pointers to an opaque structure can be defined

TRANSPARENT STRUCTURES: structure fully declared in the interface file, letting the client know about its fields

Abstract data types

DATA STRUCTURE: as the client, you **know** how the data is structured and you can **access the data directly** in any manner you desire

ADT: the client does **not know** how the data is structured and can only access the data through the interface provided by the ADT

COLLECTION: an ADT designed to store an arbitrary amount of data (or number of items)

- Include:
    - Stacks
    - Queues
    - Lists or sequences
    - Trees
    - Graphs
    - Sets

Stack ADT

- Items are pushed onto the top of the stack and popped off the top of the stack
- Exhibits **LIFO** behavior
- Typical operations
  - **Push:** adds an item to the top of the stack
  - **Top:** returns the item at the top of the stack
  - **Pop:** removes the item from the top of the stack and returns it
  - **Empty?:** determines if the stack is empty or not

Queue ADT

- New items are added to the back of the lines, and items are removed from the front of the line
- Exhibits **FIFO** behavior
- Typical operations:
  - **Enqueue:** adds an item to the end of the queue
  - **Front:** returns the time at the front of the queue
  - **Dequeue:** removes the item from the front of the queue and returns it
  - **Empty?:** determines if the queue is empty or not

Sequence ADT

- Useful when you want to insert, retrieve, or delete items at an arbitrary position
- **Insert-at / remove-at:** change the position of items after the insertion / removal position
- Typical operations:
  - **Length:** return the number of items in the sequence
  - **Insert:** inserts a new item at a given position
  - **At:** returns the item at a given position
  - **Remove:** removes an item at a given position and returns it

Oversized arrays

- Arrays of fixed length
- Keep track of current length and maximum length

Session 7: Efficiency

ALGORITHM: step-by-step description of how to solve a “problem”

- Not restricted to computing
- “problems “ are function descriptions (interfaces)

Algorithm comparison

- Use conservative (pessimistic) and use the worst case
- Average case running time is typically more complicated

Problems with quantifying by time

- Make year of statement
- Unit of measurement
- Machine and model (with how much memory?)
- Computer language and operating system
- Actual CPU time, or total time elapsed
- Accuracy of time measurement

Big O Notation

Constant – O(1)

- Operators, calls to simple functions

Linear – O(n)

- Simple array traversal

Quadratic – O(n^2)

- Simple array traversal with O(n) in loop body
- Nested loops

Cubic – O(n^3)

Logarithmic – O(log n)

- Fractioning data length
- Ex. for (int i = 1; i < len; i *= 2)

Logarithmic – O(n log n)

- Having a nested loop, with one of the loops as O(log n)

Exponential – $O(2^n)$
- Complex recursion

Big O arithmetic

- When adding two orders, the result is the largest of the two orders
  - Two unnested orders within a function
  - O(1) + O(1) = O(1)
  - O(1) + O(n) = O(n)
- When multiplying two orders, the result is the product of the two orders
  - One order being applied to another
  - O(log n) x O(n) = O(n log n)
  - O(1) x O(n) = O(n)

Iterative algorithms
1. Work from **innermost loop** to **outermost**
2. Determine number of iterations (in the worst case) in relation to the size of the data (n) or an outer loop counter
3. Determine running time per iteration
4. Write summation(s) and simplify the expression

Recursive algorithms
1. Identify the order of the function **excluding** recursion
2. Determine the size of the data for the next recursive call(s)
3. Write the full recurrence relation (combine step 1 & 2)
4. Look up the closed-form solution in a table
    - Recurrence relations:
      - $T(n) = O(1) + T(n - k_1) = O(n)$
      - $T(n) = O(n) + T(n - k_1) = O(n^2)$
      - $T(n) = O(n^2) + T(n - k_1) = O(n^3)$
      - $T(n) = O(1) + T(n - k_1) + T(n - k_1') = O(2^n)$
      - $T(n) = O(1) + T(n / k_2) = O(\log n)$
      - $T(n) = O(1) + k_2 * T(n / k_2) = O(n)$
      - $T(n) = O(n) + k_2 * T(n / k_2) = O(n \log n)$ where $k_0 >= 1$; $k_2 >= 2$
      - **TABLE WILL BE PROVIDED IN EXAMS**

Efficiency of sorting algorithms

- Selection sort
  - Best case : $O(n^2)$
  - Worst case : $O(n^2)$
- Insertion sort
  - Best case : $O(n)$
  - Worst case : $O(n^2)$
- Quick sort
  - Best case : $O(n \log n)$
  - Worst case : $O(n^2)$
- Binary search
  - $O(\log n)$

Session 8 : Strings

- No built-in string type
- “Convention” is an array of characters, terminated by the null-character ‘\0’
- Since strings are null-terminated, no need to pass length to functions
- Good style of have **const parameters** to communicate **no mutations** occur to the string

String functions

Strlen
- Returns the length of a string
- Time complexity : $O(n)$

Strcmp
- Compares two strings lexicographically
- Returns a negative value if str1 appears before str2, a positive value if str2 appears before str1, and 0 if both strings are equal
- Time complexity : $O(n)$

Printf
- The printf placeholder for strings is “%s”

- Time complexity : O(n), where n is the length of the string

Strcpy

- Copies the content of a string src, including the null terminator, into dst
- Can be a source of buffer overflows
    - Always ensure that dst array is large enough, including null-terminator
- Time complexity : O(n), where n is the length of src

Strcat

- Appends the content of string src onto dst
- Time complexity : O(n), where n is the length of src

Strdup

- Makes a duplicate of a string
- Similar to strcat, but allocates heap memory instead

String literals

STRING LITERAL: C strings are not initialized as an array

- For each string literal, a null-terminated const char[] is created in the global read-only section of memory
- In the code, the occurrence of the String literal is replaced with the address of the corresponding array
- Do not behave like an array
    - Content is immutable
    - Identifier is reassignable

Session 9 : Dynamic Memory

The Heap

- Memory is **allocated** from the heap upon request
- This memory s "borrows" and must be "returned" (freed) back to the heap when it is no longer needed (memory deallocation)

- If too much memory has already been allocated, attempts to borrow additional memory may fail

Advantages of the Heap

- Dynamic
  - Size of the memory to be allocated can be determined at runtime
- Resizable
  - Allocated memory can be "resized"
- Scope
  - Allocated memory persists until is it "freed"
  - A function can allocate memory that continues to be valid after the function returns
- Safety
  - If memory runs out, it can be detected and handled properly (unlike stack overflows)

Malloc

- Short **m**emory **alloc**ation
- Function which dynamically allocates memory from the heap memory section
- Declared in **<stdlib.h>**
- Use ex. struct posn *my_posn = malloc(sizeof(struct posn));
- Heap memory provided by malloc is **uninitialized**
- Should always use **sizeof** with malloc to improve portability and to improve communication
- An unsuccessful call to malloc returns NULL
  - Good style to check every malloc return value and handle a NULL instead of crashing

Free

- Every block of memory obtained through malloc must be **manually freed** before the program terminates
  - Free function deallocates the space previously allocated by malloc, calloc, or realloc
- Once a block of heap memory has been freed, reading from or writing to it is invalid and may cause errors
- Once a block of heap memory has been freed, freeing it again could cause errors
- Calling free does **not mutate** the value of a pointer
  - While the memory the pointer is pointing at has been freed and is now invalid, the pointer is still pointing at it
  - A pointer to a freed allocation is known as a **dangling pointer**
  - Sometimes advisable to assign NULL to a dangling pointer
- Run-time error occurs when calling free with memory that was not returned by malloc
- **MEMORY LEAK:** occurs when allocated heap memory is not freed before the program terminates

Realloc

- Realloc(ptr, newsize) turns a block of heap memory of newsize. If ptr is not NULL, the content of *ptr is copied over, and ptr is freed
- Preserves the contents of the old array
- The pointer returned by realloc may be the original pointer, depending on circumstances
    - Regardless only the new returned pointer can be used

Scope and side effects

- Advantage of dynamic memory is that a function can obtain memory that persists after the function has returned
- Allocating (and deallocating) memory has a side effect: **modifies the "state" of the heap**
    - Must be documented
    - Ex. "effects: allocates heap memory [caller must free]"
- Inversely, a function could free memory it did not allocate
    - Side effect: "effects: data becomes invalid"

Merge Sort

1. The array is split (in half) into two separate arrays
2. The two arrays are sorted and then merged back together into the original array
3. Uses helper function "merge"

```
// merge(dest, src1, len1, src2, len2) modifies dest[] to contain
//   the elements from both src1[] and src2[] in sorted order.
// requires: length of dest[] is at least len1 + len2 [not asserted]
//           src1[] and src2[] are sorted [not asserted]
// effects:  modifies dest[]
// time:     O(n), where n is len1 + len2
void merge(int dest[], const int src1[], int len1,
                       const int src2[], int len2) {
  int pos1 = 0;
  int pos2 = 0;
  for (int i = 0; i < len1 + len2; ++i) {
    if (pos1 == len1 || (pos2 < len2 && src2[pos2] < src1[pos1])) {
      // taking data from src2
      dest[i] = src2[pos2];
      ++pos2;
    } else {
      // taking data from src1
      dest[i] = src1[pos1];
      ++pos1;
    }
  }
}
```

```
void merge_sort(int arr[], int arr_len) {
  if (arr_len > 1) { // recursive condition
    // splitting arr into two sub-arrays of equal length (+/- 1)
    int len_left = arr_len / 2;          // left half of arr
    int len_right = arr_len - len_left; // right half of arr
    int *arr_left = malloc(len_left * sizeof(int));
    int *arr_right = malloc(len_right * sizeof(int));
    for (int i = 0; i < len_left; ++i)
      arr_left[i] = arr[i];
    for (int i = 0; i < len_right; ++i)
      arr_right[i] = arr[i + len_left];
    // sorting the two sub-arrays individually and recursively
    merge_sort(arr_left, len_left);    // sort left sub-array
    merge_sort(arr_right, len_right); // sort right sub-array
    // merging the two sorted sub-arrays back together
    merge(arr, arr_left, len_left, arr_right, len_right);
    free(arr_left);
    free(arr_right);
  }
}
```

Doubling strategy

```
if (*len_cur == len_max) {
  len_max *= 2;
  *lines = realloc(*lines, len_max * sizeof(char *));
}
```

Session 10 : Linked Data Structures

NODE: contains some data and a link to the next node in the list

- Implemented as structures (llnode)
- The link between nodes is implemented as a pointer
    - Pointer value of the last node is NULL, which indicates the end of a linked list

LINKED LIST: a sequence of nodes

- The last node in a linked list does not link to another node
- Can grow and shrink at run-time
- Significant advantage over an array is that it's possible to add and remove items from the front and middle
- Beginning of a linked list is usually implemented as a separate **wrapper structure** (llist)
    - Contains a link to the front of the linked list
- Clients interact with the linked list only through the wrapper structure llist
    - Prevents the client from directly accessing and manipulating linked data

- Llnode is a recursive data structure, whereas llist is not

Link list creation

```c
// llist.h [INTERFACE]
// ll_create() creates a new empty linked list.
// effects: allocates heap memory [client must call ll_destroy]
// time:    O(1)
struct llist *ll_create(void);

// llist.c [IMPLEMENTATION]
struct llist *ll_create(void) {
  struct llist *llst = malloc(sizeof(struct llist));
  assert(llst);
  llst->front = NULL;
  return llst;
}
```

Linked list node creation

```c
// llist.c [IMPLEMENTATION]
struct llnode *lln_create(int data) {
  struct llnode *lln = malloc(sizeof(struct llnode));
  assert(lln);
  lln->data = data;
  lln->next = NULL;
  return lln;
}
```

List insertion

Inserting at the front

```c
// llist.c [IMPLEMENTATION]
void ll_insert_front(struct llist *lst, int itm) {
  struct llnode *new_node = lln_create(itm);
  new_node->next = lst->front; // either NULL or existing node
  lst->front = new_node;
}
```

Inserting at the back

```c
// llist.c [IMPLEMENTATION]
void ll_insert_back(struct llist *lst, int itm) {
  struct llnode *new_node = lln_create(itm);
  if (lst->front == NULL) { // empty list: insert at front
    lst->front = new_node;
  } else { // non-empty list: find the node AFTER which to insert
    struct llnode *insert_after = lst->front;
    while (insert_after->next != NULL) {
      insert_after = insert_after->next;
    }
    insert_after->next = new_node;
  }
}
```

Inserting at an arbitrary position

```c
// llist.c [IMPLEMENTATION]
void slst_insert(struct llist *slst, int itm) {
  struct llnode *new_node = lln_create(itm);
  if (slst->front == NULL || itm <= slst->front->data) {
    new_node->next = slst->front;
    slst->front = new_node;
  } else {
    struct llnode *insert_after = slst->front;
    while (insert_after->next != NULL &&
           itm > insert_after->next->data) {
      insert_after = insert_after->next;
    }
    new_node->next = insert_after->next;
    insert_after->next = new_node;
  }
}
```

List traversal

```c
int ll_length(const struct llist *llst) {
  int len = 0;
  const struct llnode *current = llst->front;
  while (current) { // current != NULL
    ++len;
    current = current->next;
  }
  return len;
}
```

Node removal

Removal from the front

```c
// llist.c [IMPLEMENTATION]
void ll_remove_front(struct llist *lst) {
  assert(lst->front);
  struct llnode *to_remove = lst->front;
  lst->front = lst->front->next;
  lln_destroy(to_remove);
}
```

Removal from the back

```c
// llist.c [IMPLEMENTATION]
void ll_remove_back(struct llist *lst) {
  assert(lst->front);
  struct llnode *destroy_after = lst->front;
  struct llnode *to_destroy = lst->front;
  while (to_destroy->next != NULL) {
    destroy_after = to_destroy;
    to_destroy = to_destroy->next;
  }
  if (to_destroy == lst->front) { // remove only element
    lst->front = NULL;
  } else {                        // remove non-only element
    destroy_after->next = NULL;
  }
  lln_destroy(to_destroy);
}
```

Removal from an arbitrary position

```c
// ll_remove_item(lst, itm) removes the first occurrence of
//   item itm in list *lst and returns true if item is
//   successfully removed, and false otherwise.
bool ll_remove_item(struct llist *lst, int itm) {
  if (lst->front == NULL) return false;
  if (lst->front->data == itm) {
    ll_remove_front(lst);
    return true;
  }
  struct llnode *remove_after = lst->front;
  while (remove_after->next && itm != remove_after->next->data) {
    remove_after = remove_after->next;
  }
  if (remove_after->next == NULL) return false;
  struct llnode *to_remove = remove_after->next;
  remove_after->next = remove_after->next->next;
  lln_destroy(to_remove);
  return true;
}
```

List destruction

```
// llist.c [IMPLEMENTATION]
void ll_destroy(struct llist *lst) {
  struct llnode *current = lst->front; // basic list traversal
  while (current) {                    // basic list traversal
    struct llnode *to_destroy = current;
    current = current->next;           // basic list traversal
    lln_destroy(to_destroy);           // destroying the llnode
  }
  free(lst); // destroying the llist wrapper structure
}
```

Node augmentation

	Linked list	Linked list with back pointer	Doubly linked list with back pointer
Insert_front	O(1)	O(1)	O(1)
Insert_back	O(n)	O(1)	O(1)
Remove_front	O(1)	O(1)	O(1)
Remove_back	O(n)	O(n)	O(1)

Trees

- Nodes may have multiple children
- In a binary tree, each node has at most two children
- **ROOT NODE:** node which has no parent, whereas all others have exactly one
- **LEAF NODE:** node which has no children
- **HEIGHT:** maximum possible number of nodes from the root a leaf
    - Height of an empty tree is 0
- **NODE COUNT:** number of nodes in a tree

Binary Search Tree Implementation

Definition

```c
struct btnode {
  int data;
  struct btnode *left;
  struct btnode *right;
};

struct bst {
  struct btnode *root;
};
```

Creation

```c
// bst_create() creates a new empty BST.
// effects: allocates heap memory [client must call bst_destroy]
// time:    O(1)
struct bst *bst_create(void) {
  struct bst *bst = malloc(sizeof(struct bst));
  bst->root = NULL;
  return bst;
}
```

Traversal

```c
// bst_traverse(bst) traverses the tree bst.
// time: O(n)
void bst_traverse(const struct bst *bst) {
  assert(bst);
  if (bst->root != NULL) {
    bst_traverse_wrkr(bst->root);
  }
}

void bst_traverse_wrkr(const struct btnode *node) {
  if (node != NULL) {
    bst_traverse_wrkr(node->left);
    bst_traverse_wrkr(node->right);
  }
}
```

Node creation

```c
struct btnode *node_create(int data) {
  struct btnode *node = malloc(sizeof(struct btnode));
  node->data = data;
  node->left = NULL;
  node->right = NULL;
  return node;
}

void node_destroy(struct btnode *node) {
  assert(node);
  free(node);
}
```

Insertion

```
void bst_insert_wrkr(struct btnode *node, int data) {
  // find node after which to insert
  struct btnode *insert_after = NULL;
  while (node != NULL) {
    insert_after = node;
    if (data < node->data) {        // data should go left
      node = node->left;
    } else if (data > node->data) { // data should go right
      node = node->right;
    } else {                        // data already exists
      return;
    }
  }
  // inserting new data
  if (data < insert_after->data) {
    insert_after->left = node_create(data);
  } else if (data > insert_after->data) {
    insert_after->right = node_create(data);
  }
}
```

Trees and efficiency

- Worst case is when the tree is **unbalanced**, and every node must be visited
- Runtime of bst_insert is O(h)
  - Dependant on the height (h) rather than number of nodes (n)
  - **BALANCED TREE:** a tree with a height that is O(log n)
    - Runtime of standard tree functions is O(n)
  - **UNBALANCED TREE:** tree with a height that is not O(log n), but O(n)
  - **SELF-BALANCING TREE:** tree which **"**rearranges" the nodes to ensure the tree is always balanced

Array-based trees

- Some types of trees can be stored in arrays
  - Root is stored at index 0
  - For the node at index i:
    - Left child is stored at index 2i + 1
    - Right child is stored at index 2i + 2
    - Parent is stored at index (i - 1) / 2
  - Special sentinel value can be used to indicate an empty node (ex. NULL)

  - Tree of height h requires an array of length 2h - 1
  - An array can be re-allocated as the tree height grows

Session 11 : Generic Abstract Data Types

Void pointers

- Void pointer (void *) is the closest C has to a “generic” type
- Can store the address of **any type** of data (except functions)
- It is **not possible to dereference** void pointers
  - Address stored in a void pointer can be assigned to any pointer type variable, and then be dereferenced
  - Why malloc works for any data type

Generic functions

GENERIC FUNCTION: Functions that operate on any type of data

- Signature (void (*) (void *))
- Examples from <stdlib.h>
  - Qsort : sort an array of any type given a type-specific comparator function comp
  - Bsearch : either returns a pointer to the key in a sorted array, or NULL if not found
  - Memcpy : copies a certain amount of bytes from src to a dest

Generic ADT

- Generic container ADTS that can store any type of data by storing void pointers
- Generic ADT **does not know** the type of items it stores, and therefore **does not have** any information about the internal format of the data
  - ADT cannot perform actions that require knowledge about the internal format of the data (ex. Printing and destroying the data)
- Generally stores the data in an array of void pointers
  - I.e. void **
  - **Void ** is not generic**; it is a pointer to a void * and therefore can be dereferenced and behaves like any other array

Design decisions

- **Array** : for frequently accessing elements at **specific positions** (random access)
- **Linked list** : for **sequenced data** if frequently adding and removing elements at the start
- **Self-balancing BST** : for **unsequenced data** if frequently searching for, adding, and removing items
- **Sorted array** : for rarely adding and removing elements, but **frequently searching** for elements and selecting the data in **sorted order**