(3-2) Basics of a Stack

Instructor - Andrew S. O'Fallon
CptS 122 (January 26, 2018)
Washington State University

WASHINGTON STATE UNIVERSITY
World Class. Face to Face.

What is a Stack?

- A finite sequence of nodes, where only the top node may be accessed
- Insertions (PUSHes) may only be made at the top and deletions (POPs) may only be made at the top
  - A stack is referred to as a last-in, first-out (LIFO) data structure
  - Consider a pile or “stack” of plates; as you unload your dishwasher, the most recent plate is placed on top of the last plate, etc.; as you need a plate, you grab one from the top of the stack
- A stack is a restricted or constrained list
- We will focus most of our attention on linked list implementations of stacks

The Function-Call Stack (1)

- Refer to D & D Section 6.11
- We are aware of the function call stack; it is LIFO
- Also known as the *program-execution* stack, *run-time* stack, *program* stack, or simply "the *stack*"
- Works behind the scenes – supports the function call/return *mechanism – LIFO*
  - Necessary to track sequence of called functions

The Function-Call Stack (2)

- Supports the *creation*, *maintenance*, and *destruction* of each called function's *local* variables
- Call stack memory is placed in RAM; monitored closely by CPU

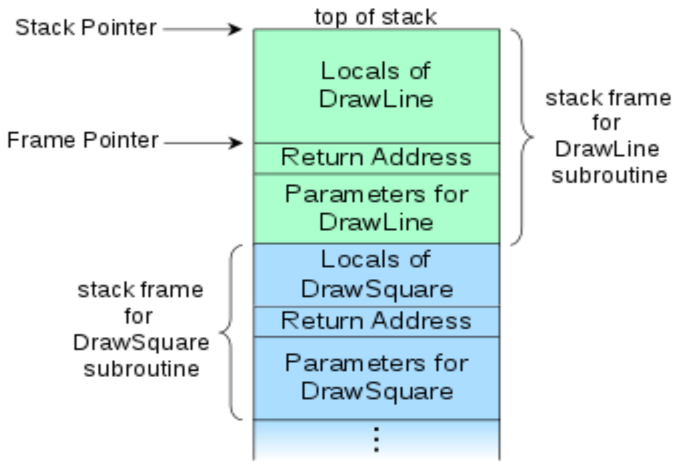
The Function-Call Stack (3)

- When a function declares a variable, it is “pushed” onto the stack (dynamic memory is not though!)
- Parameters are also passed using the call stack

A. O’Fallon, J. Hagemeister

The Function-Call Stack (4)

- How to use the call stack when debugging in MS VS 2015: https://msdn.microsoft.com/en-us/library/a3694ts5.aspx
- Diagram of call stack - courtesy of https://en.wikipedia.org/wiki/Call_stack



A. O'Fallon, J. Hagemeister

Stack Frames (1)

- Each *called* function must eventually return control to the *calling* function

```
void function1(void) // calling function
{
        function2(); // called function
        // after executing function2(),
        // control returns back to function1()
}
```

- The system must track the *return address* that each called function needs to return control to the calling function – the *function-call* stack handles this info

A. O’Fallon, J. Hagemeister

Stack Frames (2)

- Each time a function *calls* another function, an entry is *pushed* to the stack
  - The entry is called the *stack frame* or *activation record*, which contains the return address required for the called function to return to the calling function
  - The entry also contains some other information discussed later

Stack Frames (3)

- If called function returns, instead of calling another function before returning, then the stack frame for the function call is *popped*, and control transfers to the *return address* in the stack frame
- The information required for the *called* function to return to its caller is always at the *top* of the call stack!

A. O'Fallon, J. Hagemeister

Stack Frames (4)

- If a called function makes a call to *another* function, then the *stack frame* for the new function is *pushed* to the top of the stack

A. O'Fallon, J. Hagemeister

Stack Frames and Local Variables (1)

- Local variables including parameters and variables declared by the function are reserved in the stack frame
  - The reason is these variables need to remain active if a function makes a call to another function and "go away" when the function *returns* to its caller

A. O'Fallon, J. Hagemeister

Stack Frames and Local Variables (2)

- Stack Overflow
  - If more function calls occur than can be handled by the finite amount of memory for the function call-stack, then an error called *stack overflow* occurs
  - There is high potential for this occurring with recursion, on problems that require a lot of recursive steps!

A. O'Fallon, J. Hagemeister

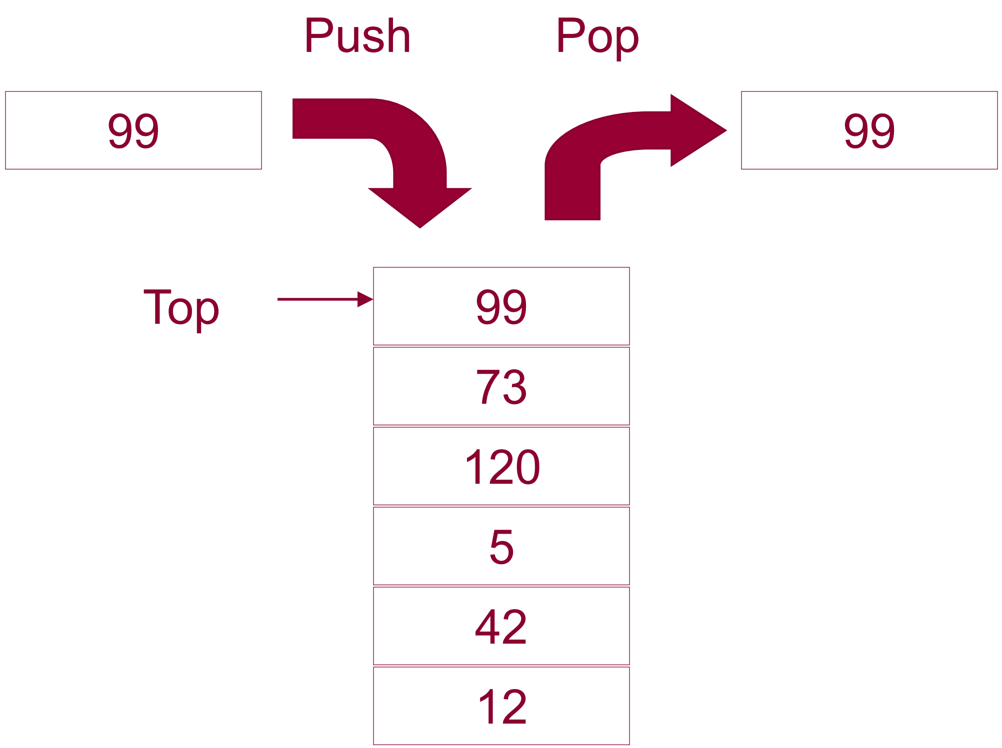
Video Explanation of Call Stack

- https://www.youtube.com/watch?v=Q2sFmqvpBe0

The Heap

- A region of memory that is not managed for you (unlike with the stack)
- We need to explicitly deallocate (free) the memory

Typical Representation of Stack of Integers



A. O'Fallon, J. Hagemeister

Struct StackNode

- For these examples, we'll use the following definition for `stackNode`:

```
typedef struct stackNode
{
    char data;
    // self-referential
    struct stackNode *pNext;
} StackNode;
```

Initializing a Stack (1)

- **InitStack (S)** Procedure to initialize the stack S to empty
- Our implementation:

```
void initStack (StackNode **pStack)
{
    // Recall: we must dereference a
    // pointer to retain changes
    *pStack = NULL;
}
```


A. O'Fallon, J. Hagemeister

Initializing a Stack (2)

- The `initStack()` function is elementary and is not always implemented
- We may instead initialize the pointer to the top of the stack with `NULL` within `main()`

```
int main (void)
{
    StackNode *pStack = NULL; // points to
                              // stack top
    …
}
```


A. O'Fallon, J. Hagemeister

Checking for Empty Stack (1)

- **StackIsEmpty (L) -> b:** Boolean function to return TRUE if S is empty
- Our implementation:

```
int isEmpty (StackNode *pStack)
{
    int status = 0; // False initially

    if (pStack == NULL) // The stack is empty
    {
        status = 1; // True
    }

    return status;
}
```

Checking for Empty Stack (2)

- Note: we could substitute the `int` return type with an enumerated type such as Boolean

```
typedef enum boolean
{
    FALSE, TRUE
} Boolean;
```


A. O'Fallon, J. Hagemeister

Checking for Empty Stack (3)

- Our implementation with `Boolean` defined:

```
Boolean isEmpty (StackNode *pStack)
{
    Boolean status = FALSE;

    if (pStack == NULL)
    {
        status = TRUE;
    }

    return status;
}
```


A. O'Fallon, J. Hagemeister

Printing Data in Stack (1)

- Our implementation:

```
void printStackIterative (StackNode *pStack)
{
    printf (“X -> “);
    while (!isEmpty (pStack))
    {
        printf (“%c -> “, pStack -> data);
        // Get to the next item
        pStack = pStack -> pNext;
    }
    printf (“NULL\n”);
}
```

Printing Data in Stack (2)

- Another possible implementation using recursion:

```
void printStackRecursive (StackNode *pStack)
{
	if (!isEmpty (pStack)) // Recursive step
	{
		printf (“| %c |\n”, pStack -> data);
		printf (“   |  \n”); // Trying to imitate link
		printf (“   V  \n”);
		// Get to the next item
		pStack = pStack -> pNext;
		printStackRecursive (pStack);
	}
	else // Base case
	{
		printf (“NULL\n”);
	}
}
```


A. O’Fallon, J. Hagemeister

Inserting Data into a Stack

- **Push (S,e):** Procedure to insert a node with information e into S; in case S is empty, make a node containing e the only node in S and the current node
- Please consider these basic specifications for stack operations in the future; However, I will only show code from this point forward

Inserting Data onto Top of Stack w/o Error Checking (1)

- Our implementation:

```
void push (StackNode **pStack, char newData)
{
	StackNode *pMem = NULL;

	pMem = (StackNode *) malloc (sizeof (StackNode));
	// Initialize the dynamic memory
	pMem -> data = newData;
	pMem -> pNext = NULL;

	// Insert the new node onto top of stack
	pMem -> pNext = *pStack;
	*pStack = pMem;

}
```

- Does this look similar to insertAtFront () for a linked list? Yes!!!!!!

A. O’Fallon, J. Hagemeister

Inserting Data onto Top of Stack w/o Error Checking (2)

- Let’s define a new function which handles the dynamic allocation and initialization of a node:

```
StackNode * makeNode (char newData)
{
    StackNode *pMem = NULL;

    pMem = (StackNode *) malloc (sizeof (StackNode));
    // Initialize the dynamic memory
    pMem -> data = newData;
    pMem -> pNext = NULL;

    return pMem;
}
```


A. O’Fallon, J. Hagemeister

Inserting Data onto Top of Stack w/o Error Checking (3)

- Now we can reorganize our code and take advantage of the new function:

```
void push (StackNode **pStack, char newData)
{
    StackNode *pMem = NULL;

    pMem = makeNode (newData);

    // Insert the new node onto top of stack
    pMem -> pNext = *pStack;
    *pStack = pMem;
}
```

Inserting Data onto Top of Stack with Error Checking (1)

- Let's modify our code so that we can check for dynamic memory allocation errors
- We'll start with `makeNode()`:

```
StackNode * makeNode (char newData)
{
      StackNode *pMem = NULL;

      pMem = (StackNode *) malloc (sizeof (StackNode));
      if (pMem != NULL)
      {
              // Initialize the dynamic memory
              pMem -> data = newData;
              pMem -> pNext = NULL;
      }
      // Otherwise no memory is available; could use else, but
      // it's not necessary

      return pMem;
}
```


A. O'Fallon, J. Hagemeister

Inserting Data onto Top of Stack with Error Checking (2)

- Let's define a `Boolean` enumerated type as follows:

```
typedef enum boolean
{
      FALSE, TRUE
} Boolean; // To be used to indicate success of push ()
```

- Now let's add some error checking to `push()`:

```
Boolean push (StackNode **pStack, char newData)
{
        StackNode *pMem = NULL;
        Boolean status = FALSE; // Assume can't insert a new node; out of memory

        pMem = makeNode (newData);

        if (pMem != NULL) // Memory was available
        {
                // Insert the new node onto top of stack
                pMem -> pNext = *pStack;
                *pStack = pMem;
                status = TRUE; // Successfully added a node to the stack!
        }

        return status;
}
```


A. O'Fallon, J. Hagemeister

Removing Data from Top of Stack (1)

- We will sometimes apply defensive design practices and ensure the stack is not empty; if we do not, then the precondition that must be satisfied is that the stack is not empty!
- This implementation of `pop()` checks for removal errors and doesn't return the data popped from the stack:

```
Boolean pop(StackNode **pStack)
{
	Boolean status = FALSE;
	StackNode *pTop = NULL;

	if (!isEmpty (*pStack)) // Stack is not empty; defensive design
	{
		pTop = *pStack; // Temp storage of top of stack
		*pStack = (*pStack)->pNext;
		free (pTop); // Remove the top node
		status = TRUE; // Successfully removed the top node
	}

	return status;
}
```


A. O'Fallon, J. Hagemeister

Removing Data from Top of Stack (2)

- This implementation of `pop()` returns the data removed from the top of the stack

```
char pop(StackNode **pStack)
{
      StackNode *pTop = NULL;
      character retData = '\0';

      if (!isEmpty (*pStack)) // Stack is not empty; defensive design
      {
            pTop = *pStack; // Temp storage of top of stack
            retData = (*pStack) -> data; // Keep data in top node
            *pStack = (*pStack) -> pNext;
            free (pTop); // Remove the top node
      }

      return retData;
}
```


A. O'Fallon, J. Hagemeister

Retrieving Data from Top of Stack w/o Deleting Nodes

- The `peek()` or `top()` function does not modify the stack; it just returns the data in the top of the stack (it "peeks" at the data)

```
char peek (StackNode *pStack)
{
     character retData = '\0';

     if (!isEmpty (pStack)) // Stack is not empty; defensive design
     {
          retData = pStack -> data;
     }

     return retData;
}
```


A. O'Fallon, J. Hagemeister

Stack Applications

- Reversing strings
- Checking for palindromes
- Searching for a path in a maze
- Tower of Hanoi
- Evaluating infix expressions
- Function call stacks
- Many others…

A. O'Fallon, J. Hagemeister

Closing Thoughts

- Can you build a driver program to test these functions?
- `push()` for a stack is essentially the same operation as `insertFront()` for a linked list…
- `pop()` is `deleteFront()` for a linked list
- If you know how to implement a linked list you should be able to implement a stack…
- You can implement a stack without using links; Hence, you can use an array as the underlying structure for the stack
- Continue to discuss why you would use a dynamic linked list instead of a dynamic linked stack and vice versa

A. O'Fallon, J. Hagemeister

Next Lecture…

- Queues

References

- P.J. Deitel & H.M. Deitel, *C: How to Program* (8th ed.), Prentice Hall, 2016
- J.R. Hanly & E.B. Koffman, *Problem Solving and Program Design in C (7th Ed.)*, Addison-Wesley, 2013

Collaborators

- Jack Hagemeister

(2-1) Data Structures & The Basics of a Linked List I

Instructor - Andrew S. O'Fallon
CptS 122 (January 17, 2018)
Washington State University

How do we Select a Data Structure? (1)

- Select a data structure as follows:
  - Analyze the problem and requirements to determine the resource constraints for the solution
  - Determine basic operations that must be supported
    - Quantify resource constraints for each operation
  - Select the data structure that best fits these requirements/constraints

- Courtesy of Will Thacker, Winthrop University

How do we Select a Data Structure? (2)

- Questions that must be considered:
  - Is the data inserted into the structure at the beginning or the end? Or are insertions interspersed with other operations?
  - Can data be deleted?
  - Is the data processed in some well-defined order, or is random access allowed?

- Courtesy of Will Thacker, Winthrop University

Other Considerations for Data Structures? (1)

- Each data structure has costs and benefits
- Rarely is one data structure better than another in all situations
- A data structure requires:
  - Space for each data item it stores,
  - Time to perform each basic operation,
  - Programming effort

- Courtesy of Will Thacker, Winthrop University

Other Considerations for Data Structures? (2)

- Each problem has constraints on available time and space
- Only after a careful analysis of problem characteristics can we know the best data structure for the task

- Courtesy of Will Thacker, Winthrop University

Definition of Linked List

- A finite sequence of nodes, where each node may be only accessed sequentially (through links or pointers), starting from the first node
- It is also defined as a linear collection of self-referential structures connected by pointers

Conventions

- An uppercase first character of a function name indicates that we are referencing the List ADT operation
- A lowercase first character of a function indicates our implementation

A. O'Fallon, J. Hagemeister

Struct Node

- For these examples, we'll use the following definition for `Node`:

```
typedef struct node
{
    char data;
    // self-referential
    struct node *pNext;
} Node;
```


A. O'Fallon, J. Hagemeister

Initializing a List (1)

- **InitList (L)** Procedure to initialize the list L to empty
- Our implementation:

```
void initList (Node **pList)
{
    // Recall: we must dereference a
    // pointer to retain changes
    *pList = NULL;
}
```


A. O'Fallon, J. Hagemeister

Initializing a List (2)

- The `initList()` function is elementary and is not always implemented
- We may instead initialize the pointer to the start of the list with `NULL` within `main()`

```
int main (void)
{
    Node *pList = NULL;
    …
}
```


A. O'Fallon, J. Hagemeister

Checking for Empty List (1)

- **ListIsEmpty (L) -> b:** Boolean function to return TRUE if L is empty
- Our implementation:

```
int isEmpty (Node *pList)
{
    int status = 0; // False initially

    if (pList == NULL) // The list is empty
    {
        status = 1; // True
    }

    return status;
}
```

Checking for Empty List (2)

- Note: we could substitute the `int` return type with an enumerated type such as Boolean

```
typedef enum boolean
{
    FALSE, TRUE
} Boolean;
```


A. O'Fallon, J. Hagemeister

Checking for Empty List (3)

- Our implementation with `Boolean` defined:

```
Boolean isEmpty (Node *pList)
{
     Boolean status = FALSE;

     if (pList == NULL)
     {
          status = TRUE;
     }

     return status;
}
```


A. O'Fallon, J. Hagemeister

Printing Data in List (1)

- Our implementation:

```
void printListIterative (Node *pList)
{
     printf ("X -> ");
     while (pList != NULL)
     {
          printf ("%c -> ", pList -> data);
          // Get to the next item
          pList = pList -> pNext;
     }
     printf ("NULL\n");
}
```


A. O'Fallon, J. Hagemeister

Printing Data in List (2)

- Another possible implementation using `isEmpty():`

```
void printListIterative (Node *pList)
{
    printf ("X -> ");
    while (!isEmpty (pList))
    {
        printf ("%c -> ", pList -> data);
        // Get to the next item
        pList = pList -> pNext;
    }
    printf ("NULL\n");
}
```

Printing Data in List (3)

- We can determine the end of the list by searching for the `NULL` pointer
- If the list is initially empty, no problem, the `while()` loop will not execute

A. O'Fallon, J. Hagemeister

Inserting Data at Front of List

- **InsertFront (L,e):** Procedure to insert a node with information e into L as the first node in the List; in case L is empty, make a node containing e the only node in L and the current node

A. O'Fallon, J. Hagemeister

Inserting Data at Front of List w/o Error Checking (1)

- Our implementation:

```
void insertFront (Node **pList, char newData)
{
     Node *pMem = NULL;

     pMem = (Node *) malloc (sizeof (Node));
     // Initialize the dynamic memory
     pMem -> data = newData;
     pMem -> pNext = NULL;

     // Insert the new node into front of list
     pMem -> pNext = *pList;
     *pList = pMem;

}
```


A. O'Fallon, J. Hagemeister

Inserting Data at Front of List w/o Error Checking (2)

- Let's define a new function which handles the dynamic allocation and initialization of a node:

```
Node * makeNode (char newData)
{
     Node *pMem = NULL;

     pMem = (Node *) malloc (sizeof (Node));
     // Initialize the dynamic memory
     pMem -> data = newData;
     pMem -> pNext = NULL;

     return pMem;
}
```


A. O'Fallon, J. Hagemeister

Inserting Data at Front of List w/o Error Checking (3)

- Now we can reorganize our code and take advantage of the new function:

```
void insertFront (Node **pList, char newData)
{
     Node *pMem = NULL;

     pMem = makeNode (newData);

     // Insert the new node into front of list
     pMem -> pNext = *pList;
     *pList = pMem;

}
```

Inserting Data at Front of List w/ Error Checking (1)

- Let's modify our code so that we can check for dynamic memory allocation errors
- We'll start with `makeNode ()`:

```
Node * makeNode (char newData)
{
     Node *pMem = NULL;

     pMem = (Node *) malloc (sizeof (Node));
     if (pMem != NULL)
     {
             // Initialize the dynamic memory
             pMem -> data = newData;
             pMem -> pNext = NULL;
     }
     // Otherwise no memory is available; could use else, but
     // it's not necessary

     return pMem;
}
```


A. O'Fallon, J. Hagemeister

Inserting Data at Front of List w/ Error Checking (2)

- Now let's add some error checking to `insertFront()`:

```
void insertFront (Node **pList, char newData)
{
      Node *pMem = NULL;

      pMem = makeNode (newData);

      if (pMem != NULL) // Memory was available
      {
            // Insert the new node into front of list
            pMem -> pNext = *pList;
            *pList = pMem;
      }
      else // Can't allocate anymore dynamic memory
      {
          printf ("WARNING: No memory is available for data insertion!\n")
      }

}
```

Closing Thoughts

- Can you build a driver program to test these functions?
- Is it possible to return a `Boolean` for `insertFront()` to indicate a memory allocation error, where TRUE means error and FALSE means no error?
- `insertFront()` will be seen again with a Stack data structure…

Next Lecture…

- Continue our discussion and implementation of linked lists

A. O'Fallon, J. Hagemeister

References

- P.J. Deitel & H.M. Deitel, *C: How to Program* (8th ed.), Prentice Hall, 2017
- J.R. Hanly & E.B. Koffman, *Problem Solving and Program Design in C (7th Ed.)*, Addison-Wesley, 2013

A. O'Fallon, J. Hagemeister

Collaborators

- Jack Hagemeister

(1 - 2) Introduction to C Data Structures & Abstract Data Types

Instructor - Andrew S. O'Fallon
CptS 122 (January 12, 2018)
Washington State University

WASHINGTON STATE UNIVERSITY
World Class. Face to Face.

What is a Data Structure?

- A software construct describing the organization and storage of information
  - Designed to be accessed efficiently
  - Composite of related items
- An implementation of an abstract data type (ADTs) to be defined later
- Defined and applied for particular applications and/or tasks

Data Structures Exposed

- You’ve already seen a few fixed-sized data structures
  - Arrays
  - Structures or structs in C

A. O’Fallon, J. Hagemeister

Review of Basic C Data Structures (1)

- Recall an *array* is a collection of related data items
  - Accessed by the same variable name and an index
  - Data is of the same type
  - Items are contiguous in memory
  - Subscripts or indices must be integral and 0 or positive only
- Our visual representation of an array of chars, where first row is index and second is contents

index	0	…	n-2	n-1
contents	‘b’	…	‘3’	‘\0’

Review of Basic C Data Structures (2)

- Recall a *structure* or *struct* is a collection of related fields or variables under one name
  - Represent real world objects
  - Each field may be of a different data type
  - The fields are contiguous in memory
- Example struct describing a dog

```
typedef struct dog
{
    char *breed; // need to allocate memory for
                 // string separately
    char name[100]; // memory is included for string
    double weight;
} Dog;
```


A. O'Fallon, J. Hagemeister

How Can We Expand on Our Data Structure Knowledge?

- In this course we will focus on dynamic data structures
  - These grow and shrink at runtime
- The major dynamic data structures include:
  - Lists
  - Stacks
  - Queues
  - Binary Trees
  - Binary Search Trees (BSTs)

A. O'Fallon, J. Hagemeister

Basic Applications of Dynamic Data Structures (1)

- Lists are collections of data items lined up in a row
  - Insertions and deletions may be made anywhere
  - May represent movie & music collections, grocery store lists, & many more…
- Stacks are restricted lists
  - Insertions and deletions may be made at one end only
    - These are Last In, First Out (LIFO) structures
  - May be used with compilers & operating systems, & many more applications…

A. O'Fallon, J. Hagemeister

Basic Applications of Dynamic Data Structures (2)

- Queues are also restricted lists
  - Insertions are made at the back of the queue and deletions are made from the front
    - These are First In, First Out (FIFO) structures
  - May represent waiting lines, etc.
- BSTs require linked data items
  - Efficient for searching and sorting of data
  - May represent directories on a file system, etc.
- This course will focus on these dynamic data structures and corresponding implementations in both C and C++

A. O'Fallon, J. Hagemeister

What do these C Dynamic Structures have in Common?

- Of course dynamic growing and shrinking properties…
- Implemented with pointers
  - Recall a *pointer* is a variable that stores as its contents the address of another variable
    - Operators applied to pointers include
      - Pointer declaration – i.e. char *ptr
      - Dereference or indirection – i.e. *ptr
      - Address of – i.e. &ptr
      - Assignment – i.e. ptr1 = ptr2
      - Others?
- Require the use of structs
  - Actually self-referential structures for linked implementations

A. O'Fallon, J. Hagemeister

What is a Self-Referential Structure?

- A struct which contains a pointer field that represents an address of a struct of the same type
- Example

```
typedef struct node
{
    char data;
    // self-referential
    struct node *pNext;
} Node;
```

Dynamic Memory Allocation / De-allocation in C (1)

- The growing and shrinking properties may be achieved through functions located in <stdlib.h> including:
  - `malloc()` for allocating/growing memory
  - `free()` for de-allocating/shrinking memory
  - `realloc()` for resizing memory
  - Also consider `calloc()`

A. O'Fallon, J. Hagemeister

Dynamic Memory Allocation / De-allocation in C (2)

- Assume the following:

```
Node *pItem = NULL;
```

- How to use `malloc()`

```
pItem = (Node *) malloc (sizeof (Node));
// Recall malloc ( ) returns a void *,
// which should be typecasted
```

- How to use `free()`

```
free (pItem);
// Requires the pointer to the memory to be
// de-allocated
```

- How to use `realloc()`

```
pItem = realloc (pItem, sizeof (Node) * 2);
// Allocates space for two Nodes and
// returns pointer to beginning of resized
// memory
```


A. O'Fallon, J. Hagemeister

How Do We Know Which Values and Operations are Supported?

- Each data structure has a corresponding model represented by the abstract data type (ADT)
  - The model defines the behavior of operations, but not how they should be implemented

A. O'Fallon, J. Hagemeister

Abstract Data Types

- Abstract Data Types or ADTs according to National Institute of Standards and Technology (NIST)
  - Definition: *A set of data values and associated operations that are precisely specified independent of any particular implementation.*

A. O'Fallon, J. Hagemeister

Data Structure

- Data Structures according to NIST
  - Definition: *An organization of information, usually in memory, for better algorithm efficiency, such as queue, stack, linked list, heap, dictionary, and tree, or conceptual unity, such as the name and address of a person. It may include redundant information, such as length of the list or number of nodes in a subtree.*

ADTs versus Data Structures

- Many people think that ADTs and Data Structures are interchangeable in meaning
  - ADTs are logical descriptions or specifications of data and operations
    - To abstract is to leave out concrete details
  - Data structures are the actual representations of data and operations, i.e. implementation
- Semantic versus syntactic

A. O'Fallon, J. Hagemeister

Specification of ADT

- Consists of at least 5 items
  - Types/Data
  - Functions/Methods/Operations
  - Axioms
  - Preconditions
  - Postconditions
  - Others?

Example Specification of List ADT (1)

- Description: A list is a finite sequence of nodes, where each node may be only accessed sequentially, starting from the first node
- Types/Data
  - e is the element type
  - L is the list type

A. O'Fallon, J. Hagemeister

Example Specification of List ADT (2)

- Functions/Methods/Operations
  - **InitList (L):** Procedure to initialize the list L to empty
  - **DestroyList (L):** Procedure to make an existing list L empty
  - **ListIsEmpty (L) -> b:** Boolean function to return TRUE if L is empty
  - **ListIsFull (L) -> b:** Boolean function to return TRUE if L is full
  - **CurIsEmpty (L) -> b:** Boolean function to return TRUE if the current position in L is empty

Example Specification of List ADT (3)

- Functions/Methods/Operations Continued
  - **ToFirst (L):** Procedure to make the current node the first node in L; if the list is empty, the current position remains empty
  - **AtFirst (L) -> b:** Boolean function to return TRUE if the current node is the first node in the list or if the list and the current position are both empty
  - **AtEnd (L) -> b:** Boolean function to return TRUE if the current node is the last node in the list or if the list and the current position are both empty
  - **Advance (L):** Procedure to make the current position indicate the next node in L; if the current node is the last node the current position becomes empty

A. O'Fallon, J. Hagemeister

Example Specification of List ADT (4)

- Functions/Methods/Operations Continued Again
  - **Insert (L,e):** Procedure to insert a node with information e before the current position or, in case L was empty, as the only node in L; the new node becomes the current node
  - **InsertAfter (L,e):** Procedure to insert a node with information e into L after the current node without changing the current position; in case L is empty, make a node containing e the only node in L and the current node
  - **InsertFront (L,e):** Procedure to insert a node with information e into L as the first node in the List; in case L is empty, make a node containing e the only node in L and the current node
  - **InsertInOrder (L,e):** Procedure to insert a node with information e into L as node in the List, order of the elements is preserved; in case L is empty, make a node containing e the only node in L and the current node

A. O'Fallon, J. Hagemeister

Example Specification of List ADT (5)

- Functions/Methods/Operations Continued One Last Time
  - **Delete (L):** Procedure to delete the current node in L and to have the current position indicate the next node; if the current node is the last node the current position becomes empty
  - **StoreInfo (L,e):** Procedure to update the information portion of the current node to contain e; assume the current position is nonempty
  - **RetrieveInfo (L) -> e:** Function to return the information in the current node; assume the current position is nonempty

A. O’Fallon, J. Hagemeister

Example Specification of List ADT (6)

- Axioms
  - Empty ()?
  - Not empty ()?
  - Others?
- Preconditions
  - Delete () requires that the list is not empty ()
- Postconditions
  - After Insert () is executed the list is not empty ()
- Others?

A. O'Fallon, J. Hagemeister

Visual of List ADT

- View diagrams on the board
  - Nodes?
  - List?

Next Lecture…

- Introduction to implementation of a dynamically linked list

References

- P.J. Deitel & H.M. Deitel, *C++: How to Program* (10th ed.), Pearson Education Inc, 2017
- J.R. Hanly & E.B. Koffman, *Problem Solving and Program Design in C (7th Ed.)*, Addison-Wesley, 2013

Collaborators

- Jack Hagemeister

(1-1) C Review: Pointers, Arrays, Strings, & Structs

Instructor - Andrew S. O'Fallon
CptS 122 (January 10, 2018)
Washington State University

(1-1) C Review: Pointers, Arrays, Strings, & Structs

Instructor - Andrew S. O'Fallon
CptS 122 (January 10, 2018)
Washington State University

Crash Review on Critical C Topics

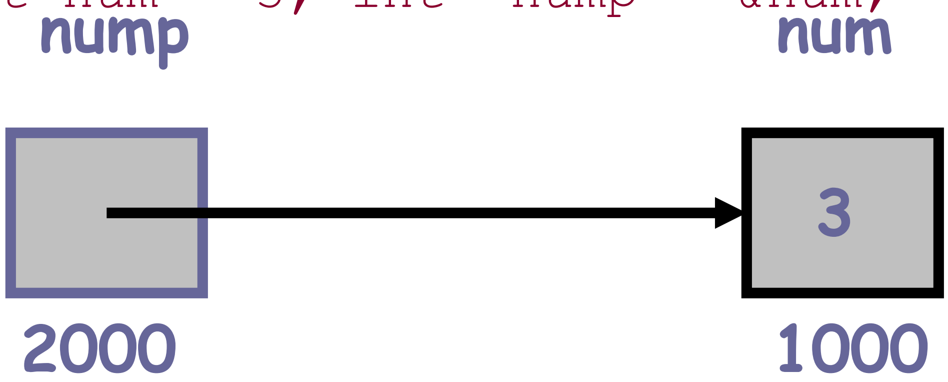
- Pointers
- Arrays
- Strings
- Structs

Pointers

C. Hundhausen, A. O'Fallon

Pointer Review (1)

- A pointer variable contains the address of another cell containing a data value
- Note that a pointer is “useless” unless we make sure that it points somewhere:
  - `int num = 3, int *nump = #`



- The *direct* value of *num* is 3, while the *direct* value of *nump* is the address (1000) of the memory cell which holds the 3

Pointer Review (2)

- The integer 3 is the *indirect* value of *nump*, this value can be accessed by following the pointer stored in *nump*
- If the indirection, dereferencing, or “pointer-following” operator is applied to a pointer variable, the indirect value of the pointer variable is accessed
- That is, if we apply **nump*, we are able to access the integer value 3
- The next slide summarizes…

Pointer Review (3)

Reference	Explanation	Value
num	Direct value of *num*	3
nump	Direct value of *nump*	1000
**nump*	Indirect value of *nump*	3
&*nump*	Address of *nump*	2000

C. Hundhausen, A. O'Fallon

Pointers as Function Parameters (1)

- Recall that we define an output parameter to a function by passing the address (&) of the variable to the function
- The output parameter is defined as a pointer in the formal parameter list
- Also, recall that output parameters allow us to return more than one value from a function
- The next slide shows a long division function which uses *quotientp* and *remainderp* as pointers

Pointers as Function Parameters (2)

- Function with Pointers as Output Parameters

```
#include <stdio.h>

void long_division (int dividend, int divisor, int *quotientp, int *remainderp);

int main (void)
{
        int quot, rem;

        long_division (40, 3, &quot, &rem);
        printf ("40 divided by 3 yields quotient %d ", quot);
        printf ("and remainder %d\n", rem);

        return 0;
}

void long_division (int dividend, int divisor, int *quotientp, int *remainderp)
{
        *quotientp = dividend / divisor;
        *remainderp = dividend % divisor;
}
```

Arrays

What is an array?

- A sequence of items that are contiguously allocated in memory
- All items in the array are of the same data type and of the same size
- All items are accessed by the same name, but a different index
- The length or size is fixed

C. Hundhausen, A. O'Fallon

More About Arrays

- An array is a data structure
  - A data structure is a way of storing and organizing data in memory so that it may be accessed and manipulated efficiently

Uses for Arrays?

- Store related information
  - Student ID numbers
  - Names of players on the Seattle Seahawks roster
  - Scores for each combination in Yahtzee
  - Many more…

The Many Dimensions of an Array

- A single dimensional array is logically viewed as a linear structure
- A two dimensional array is logically viewed as a table consisting of rows and columns
- What about three, four, etc., dimensions?

C. Hundhausen, A. O'Fallon

Declaring a Single Dimensional Array (1)

- Arrays are declared in much the same way as variables:

```
int a[6];
```

declares an array `a` with 6 cells that hold integers:

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]
10	12	0	89	1	91

Notice that array indexing begins at 0.

Strings

String Fundamentals

- A string is a sequence of characters terminated by the null character ('\0')
  - "This is a string" is considered a string literal
  - A string may include letters, digits, and special characters
- A string may always be represented by a character array, but a character array is not always a string
- A string is accessed via a pointer to the first character in it

C. Hundhausen, A. O'Fallon

String Basics (1)

- As with other data types, we can even initialize a string when we declare it:

```
char name[20] = “Bill Gates";
char *name = “Bill Gates";
char name[] = {‘B’, ‘i’, ‘l’, ‘l’, ‘ ‘, ‘G’, ‘a’, ‘t’, ‘e’,
               ‘s’, ‘\0’;}
// These are equivalent string declarations!
```

- Here's what the memory allocated to `name` looks like after either of the above is executed:

null character (terminates all strings)

name	B	i	l	l		G	a	t	e	s	\0	?	?	?	?	?	?	?	?	?
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19

C. Hundhausen, A. O’Fallon

String Basics (2)

- When a variable of type `char*` is initialized with a string literal, it may be placed in memory where the string can't be modified
- If you want to ensure modifiability of a string store it into a character array when initializing it

String Basics (3)

- Arrays of Strings
  - Suppose we want to store a list of students in a class
  - We can do this by declaring an array of strings, one row for each student name:

```
#define NUM_STUDENTS 5
#define MAX_NAME_LENGTH 31
char student_names[NUM_STUDENTS][MAX_NAME_LENGTH];
```

  - We can initialize an array of strings "in line":

```
char student_names[NUM_STUDENTS][MAX_NAME_LENGTH] =
{"John Doe", "Jane Smith", "Sandra Connor", "Damien White",
 "Metilda Cougar"};
```

  - In most cases, however, we're probably going to want to read the names in from the keyboard or a file…

C. Hundhausen, A. O'Fallon

String Basics (4)

- Use `gets()` to read a complete line, including whitespace, from the keyboard until the <enter> key is pressed; the <enter> is not included as part of the string
  - Usage: `gets(my_array)`
  - If the user enters “Bill Gates” and presses <enter>, the entire string will be read into `my_array` excluding the <enter> or newline
- Use `puts()` to display a string followed by a newline
  - Usage: `puts(my_array)`

C. Hundhausen, A. O’Fallon

String Manipulation in C (1)

- Standard operators applied to most numerical (including character) types cannot be applied to strings in C
  - The assignment operator (=) can't be applied except during declaration
  - The + operator doesn't have any true meaning (in some languages it means append)
  - The relational operators (==, <, >) don't perform string comparisons
  - Others?

String Manipulation in C (2)

- The string-handling library <string.h> provides many powerful functions which may be used in place of standard operators
  - strcpy ( ) or strncpy () replaces the assignment operator
  - strcat ( ) or strncat () replaces the + or append operator
  - strcmp ( ) replaces relational operators
  - Several others…i.e. strtok ( ), strlen ( )

Pointers Representing Arrays and Strings (1)

- Consider representing two arrays as follows:
  - `double list_of_nums[20];`
  - `char your_name[40];`
- When we pass either of these arrays to functions, we use the array name without a subscript
- The array name itself represents the address of the initial array element

Pointers Representing Arrays and Strings (2)

- Hence, when we pass the array name, we are actually passing the entire array as a pointer
- So, the formal parameter for the string *name* may be declared in two ways:
  - `char name[]`
  - `char *name`
- Note that, in general, it is a good idea to pass the maximum size of the array to the function, e.g.:
  - `void func (char *name, int size);`

C. Hundhausen, A. O'Fallon

Structs

`struct` Type (1)

- C supports another kind of user-defined type: the `struct`
- `struct`s are a way to combine multiple variables into a single "package" (this is called "encapsulation")
- Sometimes referred to as an *aggregate*, where all variables are under one name
- Suppose, for example, that we want to create a database of students in a course. We could define a student `struct` as follows:

C. Hundhausen, A. O’Fallon

struct Type (2)

```
typedef enum {freshman, sophomore, junior, senior}
         class_t; /* class standing */

typedef enum {anthropology, biology, chemistry,
                english, compsci, polisci,
psychology,
                physics, engineering, sociology}
         major_t; /* representative majors */

typedef struct
  {
      int id_number;
      class_t class_standing; /* see above */
      major_t major; /* see above */
      double gpa;
      int credits_taken;
  } student_t;
```


C. Hundhausen, A. O'Fallon

`struct` Type (3)

- We can then define some students:

```
student_t student1, student2;
student1.id_num = 123456789;
student1.class_standing = freshman;
student1.major = anthropology;
student1.gpa = 3.5;
student1.credits_taken = 15;
student2.id_num = 321123456;
student2.class_standing = senior;
student2.major = biology;
studnet2.gpa = 3.2;
student2.credits_taken = 100;
```

Notice how we use the "." (selection) operator to access the "fields" of the `struct`

C. Hundhausen, A. O'Fallon

More About Structs

- Recall structs are used to represent real world objects
- They contain attributes that describe these objects
  - Such as a car, where the attributes of the struct car could include steering wheel, seats, engine, etc.
  - Such as a student, where the attributes of the struct student could include ID#, name, standing, etc.
- In many cases, we need a list or array of these objects
  - A list of cars representing a car lot
  - A list of students representing an attendance sheet

Arrays of Structs (1)

- Let's first define a struct student

```
typedef struct student
{
    int ID;
    char name[100];
    int present; // Attended class or not
} Student;
```

- Next we will build up an attendance sheet

C. Hundhausen, A. O'Fallon

Arrays of Structs (2)

```
int main (void)
{
    Student attendance_sheet[100]; // 100 students in the class

    return 0;
}
```

- Let's look at a logical view of this attendance sheet on the next slide

Arrays of Structs (3)

- Attendance sheet, which consists of multiple struct student types

0	1	2	…	99
{ID, name, present}	{ID, name, present}	{ID, name, present}	…	{ID, name, present}
1000	1108	1216		10692

C. Hundhausen, A. O'Fallon

Arrays of Structs (4)

- To initialize one item in the array, try:

  attendance_sheet[index].ID = 1111;

  strcpy (attendance_sheet[index].name, “Bill Gates”);

  Attendance_sheet[index].present = 1;

  // 1 means in attendance, 0 means not in present

Pointers to Structures

- Recall that when we have a pointer to a structure, we can use the indirect component selection operator -> to access components within the structure

```
typedef struct
{
        double x;
        double y;
} Point;

int main (void)
{
        Point p1, *struct_ptr;
        p1.x = 12.3;
        p1.y = 2.5;

        struct_ptr = &p1;

        struct_ptr->x; /* Access the x component in Point, i.e. 12.3 */
        .
        .
        .
}
```


C. Hundhausen, A. O'Fallon

Keep Reviewing C Material!

References

- J.R. Hanly & E.B. Koffman, *Problem Solving and Program Design in C (8th Ed.)*, Addison-Wesley, 2016.
- P.J. Deitel & H.M. Deitel, *C How to Program (7th Ed.)*, Pearson Education , Inc., 2013.

Collaborators

- Chris Hundhausen