1. **Intro** C is a high level language. computers perform actions on data and move it around to achieve an obj. Data is long sequences of simple elements called bits. If a specific action is to be executed, this is called **machine language**. 0s and 1s are the electrical voltage values output of a transistor. **Compiler** translates statements into machine language. RAM(main memory) is volatile and wiped when you turn off your computer. Registers are very fast, hardware mem locations where instructions are placed to be exec. **#include** directive tells the preprocessor to include the named files in the compilation process.
2. **Variable & Operators** Local Var the system “forgets” the variable once the function in where they are defined exits. **Global Var** should be defined outside of any func. They are valid everywhere in the program. Each **byte** in comp mem has an address (**addressable memory**). Variable type determines how much mem is to be set aside. **bit** = 1 binary element. **byte** = a string of 8. a **word** is a string of 16 consecutive bits. Ints are not good for math and are typically 1 byte or one word in size. **Unsigned integers** use all 8 or 16 bits in the byte or word to represent the number. **byte range** = 0-255. Word range = 0-65,535. Signed integers(default) only use 7 or 15 of the bits to represent the number. The highest bit is used to indicate the sign. High bit of 0 = pos num. High bit of 1 = neg num. byte range = -128 to 127. Word range = –32,768 to 32,767. Variable not declared = **compilation error.** 11%4 = 11 mod4 = 3(remainder). In Booleans 0 = false.
3. **Control Structures** Structured Programming: using seq, sel, and rep structures and GOTO statements. **sequential structure** is the most basic structure in computer programming, and the basis of instruction sequences. is structure = single selection struct. if/else = double selection structure. If/elseif/...else/ = multiple selection structure. **Switch** structure does the same as the multi-selection struct. You need to put a break; at the end of each case in order to end it after it has found the correct one. **Boolean operators use short circuit evaluation**. Ask questions that might otherwise cause a crash. If an **if statement has less than 2 arguments or statements you don't need a bracket**. IF-ELSE structs can be nested. break; statement exits that structure. Repetition structures = loops. **For loop**= counter control. **While loop & Do/while loop** = sentinel control. **Do/while**(exit condi.). loop continuation test is done after the body of the loop has been executed.Thus, the body of the **do/while loop is guaranteed to be executed at least once**. **Sentinel control**: The stopping criteria is set for a sentinel variable and its value is checked at every iteration. **Counter control(MOST COMMON)** means that a variable is set that counts the number of loops. When the count reaches a preset number, the loop exits. Requires a control variable be defined.Final value and increment can be mathematical functions. Counter var does not need to be in the body of the loop. **Entry condition** is that the stopping criteria is checked at the start of the loop. **Exit condition** means that the stopping criterion is checked after the loop statements are executed. **Continue;** = Causes the processor to skip the remaining statements in the loop body, but goes back to the next iteration.Typically associated with a **selection structure**. C has 7 control structures. Structures are single-entry/single-exit.
4. **User Defined Functions and Programming Styles** **Main() function is always required**. Functions can be defined and called. They must be defined before they can be called. User-defined function permits abstraction and abstraction allows top-down programming. **Functions are essentially sub-programs that carry out specific tasks. Makes the program highly reusable. CANNOT BE DEFIED WITH OTHER FUNC.** Can be and are typically defined after main(). **Func Headers: <return type> <function name>(<parameter list>)**. **Formal Parameters** – These are listed in the function definition. They act as local variables inside the function. **Actual Parameters** (arguments) – These are the actual values that are passed into the function when it is called. **When calling a function, the number of arguments that are passed must be the same as the number of formal parameters in the function definition**. Functions can be called from within other functions. IN order to call a func correctly you need the <name><precondition><post-condition> found in func prototype. inputs are called:**Parameters** when defining the function. **Arguments** when calling the function.**Call by value**: Only the value of the variable being referenced is passed, not its address. The called function cannot make any changes to the original variable.**Call by Reference**: The address of the variable being referenced is passed. The called function can now make changes to the original variable. **Prototypes are only needed for user-def functions**. Use of prototypes is **optional in C**, and required in C++. Function Identifiers are case sensitive. **Compilation errors**: These are mistakes that the compiler catches. EX: missing semicolon, undeclared var. **Runtime errors**: Your program compiles, but it terminates abnormally (crashes). EX: division by zero, infinite loops. **Logic errors**: Your program compiles and doesn't crash, but it produces incorrect output.
5. **Pointers** ***Pointer variable*** contains memory address of variable that contains values (or pointers). **Permits** creation of dynamic data structures. **Permits** dynamic allocation of memory. **Pointer** variables indirectly reference a value. **Referencing** a value through a pointer variable is called *indirection*. **Pointer** **variables** = *pointers.* **Pointers** must be declared like regular variables. **It must** be stated which type of variable they point to. **Declarations** use \*. **Pointers** assigned a value of 0 actually have the value 0 and not an address. ***Address-of operator*** (&) is a unary operator returning the address of its operand. ***Indirection operator*** (\*), or *dereferencing operator* is also unary and returns the value of the variable pointed at by the pointer. **By passing** a variable’s address to a function, we give that function the ability to modify the value of the original value. -> call by reference. **Non-constant pointer to non-constant data:** Declaration does not include *const* in any way. Data can be modified through the pointer. Pointer can be modified to point to other data. Highest level of data access to called function. This is what we have been doing up to now. int \*a; **Non-constant pointer to constant data:** Pointer can be modified to point to any data. Data that it points to cannot be modified May be used to protect the contents of a passed array. Read as “a is a pointer to an integer constant” const int \*a; **Constant pointer to non-constant data:** Pointer always points to same memory location. Data that it points to can be modified. Default value for a passed array. Pointer must be initialized when declared. Read “aptr is a constant pointer to an integer ”int x; int \* const aptr = &x; **Constant pointer to constant data:** Pointer always points to same memory location. Data that it points to cannot be modified. Read “aptr is a constant pointer to an integer constant” -  right to left Pointer arithmetic deals with manipulating the number of **memory elements** used in arithmetic int x = 5; **const int \*** const aptr = &x; **A pointer can** (++),(--), (+, +=, -, -=). **A double pointer** is a pointer to a pointer to a variable of a particular type. **Dereferencing** a double pointer results in an address. **Dereferencing** it again results in the value of the ultimate variable
6. **Arrays and Structures** **Arrays** provide space for several “variables” of the same type, ordered sequentially can be one, two or n dimensional. **The** declaration allows the compiler to set aside sufficient contiguous memory for the size of array. **Arrays** allocated statically - remain the same size throughout its scope. **Cannot** grow in size during execution.**Arrays** are passed by reference to functions and can be passed with name without brackets. **Header** and prototype indicate array is being passed. **To** traverse the length of an array, one must do it through a loop. **The** array name itself can be used directly in pointer arithmetic. **a[3] can be also referenced as \*(a+3).** a+3 could be written as &a[3]. **Pointer arithmetic** is meaningless outside of arrays. Because you cannot assume same variable type with be next to each other in memory **Structures** are a  collection of related, but dissimilar variables under one name. **The components** of a structure variables are called *members*. **They can** be accessed by the dot or arrow (for pointers) operator **Structures** can be passed to functions as: Individualstructure members. An entire structure variable. Pointer to a structure variable. Passed by value if the individual structure member or the entire structure is passed. Passed by reference if a pointer to the structure is passed.
7. **Strings and String Manipulation** In C, chars are represented by their American Standard Code for Information Interchange ASCII values. **Chars** = 8bits(1byte). Treated as integers from -128 to 127. Only some are printable (32 to 126 are normal characters).**Char arithmetic**: ‘a’+1 is ‘b’ etc. If x is lowercase then turn it upper case like so x = x – 'a' + 'A'; (subtract from itself = 0 ,then add new char)**read/write** one char at a time. **getchar()** no param, returns char entered by user. **Fgetc()** takes file pointer, returns single char from it. **Putchar()** prints single char to string. **Fputc()** takes in a char and a file pointer and prints to file..**<ctype.h>:** **Isalpha()** = letter? **Isdigit** = #? **Isalnum** = letter or #? **Isspace** = whitespace? **Isupper** = uppercase? **Islower** = lower case? **Toupper** returns upper version of a letter. **Tolower()** = returns lower **Strings**: A number of useful string functions are found in <string.h> **Reading strings**: scanf("%s", word); Caveat: When you read a string, scanf stops reading as soon as it hits whitespace **strcpy** – Copies the second string into the first string parameter (This is a good way to initialize strings) strcpy(word, "Hello"); You can think of strcpy()as =, except for strings **String Functions: strlen** – Returns an integer indicating the length of the string **strcat** – Concatenates (tacks on to the end) the second string to the first string You can think of strcat() as += except for strings **strcmp** – Compares two strings Returns 0 if the strings are the same, negative if the first comes before the second, or positive if the first comes after the second if(strcmp(word, "hello") == 0) printf("You entered hello\n"); **Comparison Operators Table**: a==b { strcmp(a,b) == 0 } || a != b  { strcmp(a,b) != 0 } || a < b  { strcmp(a,b) <  0 } || a > b  { strcmp(a,b) > 0 } || a <= b  { strcmp(a,b) <= 0 } || a >= b  { strcmp(a,b) >= 0 } || **strings**: char words [100 - # of strings][20 - max length -1 (\0)] **Printing Strings**: printf("%s", word); Precision works a little differently for strings: printf("%.6s", word); //will print AT MOST 6 chars, To access a string in an array of strings, it works the same as accessing anything else in an array,  Example: Print whatever string is in word[4] printf("%s\n", word[4]); Ex:: Read into the ith string: scanf("%s", word[i]);
8. **Advanced I/O** Inputs and outputs are performed as “streams” – sequence of characters organized into 254 char lines (inc. \n)”. 3 streams: Input-keyboard(scanf)/Output-screen(printf)/Error; streams may be re-directed to external memory etc. **Printf/scanf: %[flags][width][.precision]specifier - [optional], int Specifiers**: **%i** signed decimal integer **%u** unsigned decimal integer **%o** unsigned octal integer **%x or X** unsigned hexadecimal int (a-f or A-F) **%h** modifies the int specifier to mean short **%l** modifies the int specifier to mean long, **%E or %e** floating point value in sci-not.– e.g., 1.234567E+006 or 1.234567e+006, **%G or %g**: floats or sci-not., depends on num.. char character = ‘A’; char string[] = “This is a string”; const char \*ptr = “This is also a string”;printf(“%s\n”, “This is the first string”); ->  This is the first string; printf(“%c\n”, character); -> A; printf(“%s\n”, string); ->  This is a string; printf(“%s\n”, ptr); ->  This is also a string [.**precision]** ints ➔ printed. (Zeroes if smaller e.g. 0072). Floats: num of digits after decimal ➔ shown. Strings: chars  ➔ what is printed **Field [width]** minimum num of chars  to be printed. Does not truncate. May increase field width. shorter nums will pad with blank spaces and right justifies it **[Flags]** left justifies the num instead of right justifying, **+** shows the sign, even if positive (+5), **0**: pads with zeroes, **space**: puts a space before a positive value, **#**: prefixes an O if octal. **Advanced printf:** printf("%+05d", 47); ➔ +0047, two line printf("This is a really, really, "+ "really long string\n"); **String Output Functions:** puts() prints a string to standard output, \n, Returns an integer value putchar() prints its character argument. if a string, must be in a loop **Input Functions: scanf(format-control-string, other-arguments); %d** signed decimal integer, **%i** signed decimal, octal or hexadecimal integer (not in printf()), **%o** octal number, **%u** unsigned decimal integer, **%x/X** hexadecimal integer prefixed with x, **e, E, f, g** or **G**: floating points **%c** characters **%s** strings, **%n** num of chars in scanf() as an **int**, %p address (pointer), **%\*** character suppression. scanf() specifiers take pointers as an argument, so, the address-of an operator & must be used before the name of the variable where the value read is to be stored. **except** a string **String Input Functions:** **gets()** reads chars from input stream, ends at \n or eof; A null character (\0) is appended automatically to the end of the string. getchar() inputs next char from standard input and returns as an int, does not take an argument. Loop if string.  not auto added File i/o Basics: 1) declare a pointer of type FILE (found in stdio.h) to read/write to a file ex: FILE \*fin, \*fin1, \*fin2, \*fout; fopen( 1- string of a file name, 2- do\_what?) returns an address (fp) takes two parameters fin = fopen("test.txt", "r"); prevent r/w errors from non-existent file by: if ((fin=fopen(“results.dat”, “r”)) == NULL) printf(“file could not be opened; else {  … everything you want to do ..} File read/write an int: fscanf(fin, "%d", &x); fprintf(fout, "%d", x); “r+” – Opens (read and write), “w+” – Creates(read and write), “a+” – Append – open or create a file for updating. Close a file: fclose(fout); //OS may automatically do this, feof () == False, feof(EOF) == True, sum all ints in a file: while(!feof(fin)) { fscanf(fin, "%d ", &number);  sum += number;}
9. **Dynamic Memory Allocation** **Memory chunks** are pointed to by only a pointers. If pointer is deleted or redirected, chunk is lost. Thus, programmer needs to keep track of pointer, and free it when it’s not in use to avoid mem leaks. **Memory leak** is if you keep allocating memory and don’t free it. (like malloc() within a loop) **sizeof()** standard C function to determine how much memory a data type consumes, especially useful for structs. Sizeof() is an operator evaluated by the compiler. **malloc()** allocates uninitialized memory. Its argument is a size of chunk of memory. Returns a pointer to that chunk. Frequently used to make space for structs.   EX:  bar = malloc(sizeof(struct foo)); **free()** takes a pointer to dynamically allocated memory and gives it back to system.  EX: free(bar); **NULL** pointer value to indicate pointer isn’t pointing to anything. Trying to dereference = run time error. **NULL** functions that return pointers can return NULL to indicate failure (ex fopen). Also to indicate end of list. **allocating arrays** Arrays/pointers interchangeable. *\*bar == bar[0].  Int \*numbers; numbers = malloc(10 \*sizeof(int))*; initiatives an array of 10 int variables using malloc. **calloc()** used for allocating arrays. Takes 2 parameters. Automatically initializes mem to all 0s. *int \*numbers; numbers = calloc(10, sizeof(int));* (does same as above)
10. **Linked List** **list** sequence of elements of same type. **Main operations**: create, search/retrieve, insert, delete **contiguous list** – list as array. No pointers, easily declared/defined, random cells can be accessed because of predefined structure. Inefficient to insert things, holes left when deleted, size has to be known, finding enough contiguous free memory for large lists can be challenging **linked list** grow and shrink easily. No need to know memory needed b4 hand. Deletions don’t leave holes. Linked through pointers.  First node = head, last = tail points to NULL. Can’t access random elements they are **linear**, have to start search at the beginning. Use more memory because of pointers  **self-referencing structure** are used by linked lists. Seems illegal but C relaxes the rule that things must be defined b4 it’s used for this case.  **Traversal** of a linked list = following next pointers. Use **current** for traversal. start with current = head then current = current->next until current->next = NULL. **Create function -** simply assigns the head pointer equal to NULL. **Insert Function-** several different operations, **Insert(beginning of list)-** newnode->next = same as head | head= location of newnode, **Insert(Middle of list)-** newnode->next points to where current->next node points now | Current->next then is shifted to where newnode points **Insert(end of list)-** Set newnode->next to NULL instead of where current->next was pointing. **Delete Functions-** several different operations, **Delete(front)**- temp points to node being deleted | set head = head->next | free(temp), **Delete(middle)-** temp points to node being deleted and item before = current | current->next=temp->next | free(temp). **Delete(End)-** Same as middle except target node's next pointer = NULL
11. **Multiple File Programs** **#include “filename”** if same directory or **#include <filename>** if not in same directory **Function definitions** must be done in a single file **Function prototypes** must be included in all files where functions are called but not defined. **The identifier static** can also be used to do the same (thing that the prototype does) with functions **local variables** are also called **automatic** variables. Names can be used in different functions (confusing) **Global variables** = **external variables**. Can be identified by **extern** (in files other than the original). Declared after pre-proc directives **limit scope** of globals by using static to make them global only to functions in the same file. This is done in the function definition where global (defined in another files) is to be recognized = **internal linkage**
12. **Stacks and Queues** **Sequence**: A sequence of length 0 is empty. A sequence of length n ? 1 of elements from a set T is an ordered pair (Sn-1, t), where Sn-1 is a sequence of length n-1 of elements from T, and t is an element of T. **3 types of sequences**: (stack, queue, list). **Stacks:** (LIFO) = last-in-first-out. It specifies a certain way elements can be added to and deleted form. Element can only be added and deleted from the top.**push(item**) //func put item on top of stack **|| pop()** //func remove element at top of stack || **t top()** //func returns val at top of stack w/o removing from stack. Components**:  pop\_top()** func returns the element at the top of the stack, & deletes it from the stack**. Full()** returns T if stack full, else F**. Empty()** //returns T is stack empty, else F. **print-stack();** **Stack Phy Model:** simple to do array-based contiguous stack. Must define pointer to top. Needs to be a var to track # elements**. QUEUE Data**: (FILF) = first-in-first-out**. Entry=rear, Exit = front**. Items only add to rear. Items only taken from front. Less restrictive than stack, more than list. Components**: enqueue(item**) func to add an item at the rear position of a queue. Aka **append();** || dequeue () func that del the element at the front of the queue. AKA **serve();** || Other Functions:  **serve\_retrieve()** func which dequeues and returns value of front element removed. **Queue physical**: contiguous queue = 2 pointers. Front has 1st pointer and deletions. Rear has 2nd and additions. **Problem**: removing from front and adding to back = travel**. Queue Physical Model**: linear = useless. **Circular array**: travel in circles. Big enough array size = never run out of space. Best way for contiguous queue. **Circular Queues**: front & rear ptr's ambiguous. Empty queue and Full Queue = front & rear ptr , pointing to adj cells in array. Note: Always leave one cell in array empty- array assumed full when rear has moved within two positions of front “pointer”**Linked Stacks and Queues**:  linked stack has a ptr called top which points to the top. linked queue has two pointers, front and rear **Problems:** relative positions of the pointers can be ambiguous. –an empty queue will have the pointers pointing at adjacent cells in the array or a full queue will also have the same. Pointers can cross. **Solution**s: Leave a cell empty – array assumed full if rear is within 2 positions of front. OR keep a count of # of elements at all time. OR define a variable that becomes ‘True’ when rear moves within 1 position of front.
13. **Basic Searching and Sorting** **Internal search** = all items in electronic memory. **Static Dictionary**: No adding/removing **Dynamic**: Can add/remove **Semi-dy**: can add no rem **Records**: data we seek. **Keys**: the label of a record that is sought (don’t have to be unique) can be **compared** using operators including strcmp **Off-line** search: search requests can be edited before doing them **online**:search one by one in real time. Results of one needed b4 doing other **Sequential** **search** simplest. Only way to search **unordered** list. Searches item by item until found, or reaches end. **Best case**: 1st item (Tc x **Worst case**: Not found (N x Tc). Average: Assuming equal probability is roughly (n/2) x Tc. Tc=constant efficiency related to n **Unordered list**: More efficient to search but keeping them ordered is costly **Binary Search**: start @ the middle and split the list in half until found. (easy in arrays ,not easy in linked list). **Best case**: Tc x 1 **Worst case**: (log2 N) x Tc Average: (log2 N) x Tc/2 thus Binary search depends on: Log2 N **Sorting**: Convert unordered to ordered list. (alphabetical, reverse alphabetical, inc/dec numerical order etc) **Simple**: Insertion, Selection, Bubblesort **Complex**: Mergesort, Quicksort **Intermediate**: Shell sort Tree-based: Tree Heap **Insertion Sort**: Sorted and unsorted region. **In the Beginning**: Sorted region = first element Unsorted = everything else. **On every iteration** sorted grows by  front front to back1. Unsorted shrinks by 1 from back to front. **Complete** = unsorted is empty and entire list is sorted. for(i=1; i<n;i++) for(j=i; j>0  && x[j] < x[j-1]; j--) swap contents of x[j] and x[j-1];   (shouldn’t j = I + 1 instead?) **Memory element** = ONLY 1. The one and only list or array. **Insert efficiency** -Outer loop: Does n-1 iterations all the time. Inner  at best, 1 comparison. Worst: average size of the list n/2. **Complexity**: n(n-1)/2 roughly n^2.**Insert Complexity**: Best case: List already in order computation time = n x Tc Worse case: list in reverse order computation time = Tc x n(n-1)/

**Insertion  algorithm**

for(i=1; i<n;i++)

    for(j=i; j>0  && x[j] < x[j-1]; j--)

  swap contents of x[j] and x[j-1];

|  |  |  |  |
| --- | --- | --- | --- |
| **//Struct**  typedef struct node{    int val;    struct node \* next;  } Node;  void list\_print(Node \*);  int main(){  **Node \*curr, \*head;**    head = malloc(sizeof(Node));    head->val = 0;    curr = head;    int i;    for(i=1 ;i<=10;i++){   curr->next = malloc(sizeof(Node));   curr = curr->next;   curr->val = i;   curr->next = NULL;   }   list\_print(head);   return 0;}  void list\_print(Node \*head){    while(head !=NULL){        printf("%d\n", head->val);        head = head->next; }  }  -----------------------------------------  **//Switch Statement**  while (choice != 0){     printf("menu\n");     printf(" choise");   scanf("%d", &choice);     printf("\n");  switch(choice){    case 1:        function();         break;         default:         printf("OK, bye\n");         return 0;  }  }  ---------------------------------------  **//End of File**  while(!feof(fin)){       ptr2->next = malloc       ptr2 = ptr2->next;       fscanf(fin, "%s   %d  ", ptr2->label, &ptr2->priority);        ptr2 -> next = NULL;  }  ---------------------------------------    while (ptrs != NULL){       fprintf(fout, "%s   %d\n", ptrs->label, ptrs->priority);      ptrs = ptrs->next;  } | **void insertItem()**{     Stuffptr newnode = malloc(sizeof(Stuff));     int newPriority;     char newItem [25];     printf("\n\nInput the item and the priority: ");     scanf("%s %d", newItem, &newPriority);     strcpy(newnode->item, newItem);     newnode->priority = newPriority;     newnode->next = NULL;     Stuffptr post, prior;      if (head==NULL)            head=newnode;      else  if (newnode->priority >=            head->priority) {             newnode->next = head;             head=newnode;  }       else{             post = head;             while (post != NULL &&                    newnode->priority <=                    post->priority)                   {prior = post;                    post = post->next;  }                    newnode->next = post;                    prior->next = newnode;  }  }  ----------------------------------------------------  **void deleteItem()**  {    Stuffptr post=head, prior=head, ptr=NULL;       char deleteItem [25];       printf("\nEnter Item to delete: ");       scanf("%s", deleteItem);       if (head == NULL)           printf("The list is empty\n");      else if (strcmp(head->item,deleteItem)==0)  {        ptr=head;           head=head->next;           free(ptr);  }      else{           post=prior->next;           while (post != NULL){             if((strcmp(post->item,deleteItem)==0))                {prior->next=post->next;                  free(post);  }      else{             prior=post;             post=prior->next;   }  }  } | **//Basic Queue**  #define MAX 50  int queue\_array[MAX];  int rear = - 1;  int front = - 1;  main(){     int choice;     while (1){     printf("Menu\n");     printf("choice : ");     scanf("%d", &choice);     switch (choice){         Case x:              Function()              Break;}}}  insert(){     int add\_item;     if (rear == MAX - 1)          printf("Queue Overflow \n");      else{          if (front == - 1)          /\*If queue is initially empty \*/                   front = 0;                   printf("Inset the element in queue : ");     scanf("%d", &add\_item);     rear = rear + 1;     queue\_array[rear] = add\_item;}}  delete(){      if (front == -1||front>rear){           printf("Queue Underflow \n");           return ;}     else{          printf("Element deleted:%d\n",           queue\_array[front]);           front = front + 1;}  } /\*End of delete() \*/  **//Queue Display:**  display(){     int i;          if (front == - 1)                printf("Queue is empty \n");          else{               printf("Queue is : \n");                for (i = front; i <= rear; i++)          printf("%d ",                       queue\_array[i]);      printf("\n")  ;}} | **//Basic Stack**  #define MAXSIZE 5  Typedef struct stack{  int stk[MAXSIZE];  int top;  }STACK;  STACK s;  void push(void);  int  pop(void);  void display(void);  int main (){    int choice;    int option = 1;    s.top = -1;  printf ("STACK OPERATION\n");     while (option) {          printf ("Enter your choice\n");          scanf ("%d", &choice);      switch (choice){  }           printf (“continue(0 or 1)?\n");            scanf ("%d", &option);}}  void push (){     int num;     if (s.top == (MAXSIZE - 1)) {          printf ("Stack is Full\n");          return;}     else {          printf ("Enter the element\n");          scanf ("%d", &num);          s.top = s.top + 1;          s.stk[s.top] = num;}         return;}  **//Pop an Item:**  int pop (){     int num;     if (s.top == - 1) {         printf ("Stack is Empty\n");          return (s.top);}     else {          num = s.stk[s.top];          printf ("poped element is =          %dn",   s.stk[s.top]);          s.top = s.top - 1;}          return(num);}  void display (){      int i;      if (s.top == -1) {            printf ("Stack is empty\n");            return;}       else{            printf ("\n The status is \n");             for (i = s.top; i >= 0; i--){                  printf ("%d\n", s.stk[i]);}}                  printf ("\n");} |