Outline

- Stack
- Searching
- Sorting
- 2 dimensional array
- multi-dimensional array
- midterm review
Stack

- Data structure: data access is restricted to the top of the stack.
- push: store data
- pop: read data
- Usually implemented by a linked list, but here we use an array to mimic the function.
#define MAX_STACK_SIZE 100

void push(char stack[], char item, int *top);
{
    if (*top < MAX_STACK_SIZE) {
        ++(*top);
        stack[*top] = item;
    }
}
Pop

char pop(char stack[], int *top)
{
    char item;
    if(*top>=0) {
        item = stack[*top];
        --(*top);
    } else {
        item = STACK_EMPTY;
    }
    return item;
}
Search

- Search is a frequently used function in many applications.
- There are many ways to perform search, but the data to be searched must be organized in some way.
- Will talk about search algorithms when we discuss data structures.
Sequential Search

```c
int sequential_search(const int arr[], int n, int key)
{
    int i;
    int found=0;

    i=0;
    while(arr[i]!=key && i<n) i++;

    if (i>=n) return NOT_FOUND;
    else return i;
}
```
Sorting

- To arrange the data in some order.
- There are also many different approaches to do sorting.
- Once the data has been sorted, searching becomes trivial.
int get_min_sub(const int data[], int data_size);
int swap(int *x, int *y);

void
selection_sort(int list[], int n)
{
    int i,index_of_min;
    for (i=n; i>1; i--)
    {
        index_of_min=get_min_sub(list, n);
        swap(list[i-1],list[index_of_min]);
    }
}
Two-dimensional Arrays

Example:

```c
double a[100][50];
a[0][0] a[0][1] a[0][2] ... a[0][49]
a[1][0] ...
  ...
  ...
a[99][0] ...
```

(row)  (column)
The following expressions are equivalent:

\[
\begin{align*}
\text{int } a[2][3] &= \{1, 2, 3, 4, 5, 6\}; \\
\text{int } a[2][3] &= \{\{1, 2, 3\}, \{4, 5, 6\}\}; \\
\text{int } a[][3] &= \{\{1, 2, 3\}, \{4, 5, 6\}\}; \\
\text{int } a[][3] &= \{\{1, 2, 3\}, \{4, 5, 6\}\};
\end{align*}
\]
Pitfalls

- Is this OK?
  ```c
  int a[5]={1,2,3};
  ```

- How about this?
  ```c
  int a[5]={1,2,3,4,5};
  printf("a[5]=%d\n",a[5]);
  ```

- How about this?
  ```c
  int a[5]={1,2,3,4,5}, i=5;
  printf("a[5]=%d\n",a[i]);
  ```
Multidimensional Arrays

- The C language allows arrays of any type, including arrays of arrays.
- 1D--->2D--->3D
- Example:
  ```c
  int a[2][2][3]=
  {{1,1,0},{2,0,0}},
  {{3,0,0},{4,4,0}};
  ```
  ```c
  int a[][2][3]=
  {{1,1},{2}},
  {{3},{4,4}};
  ```
- To initialize all elements of the array to 0,
  ```c
  int a[2][2][3]=0;
  ```
Arrays and Pointers

- An array name is by itself an address, or pointer value.
- Given int a[100];
  - a[i] is equivalent to *(a+i)
- Similarly, given int *p;
  - p[i] is equivalent to *(p+i)
- The difference is that a pointer variable can take different values, while an array name is an address, or pointer value, that is fixed.
Pointer Arithmetic

- A powerful feature of C.
- If \( p \) is a pointer to a particular type, then the expression \( p+1 \) yields the correct machine address for the next variable of that type.
- Therefore, the following all make sense:
  
  \[
  p+i \\
  ++p \\
  p+=i 
  \]
The `sizeof` Operator

- How do we find out the number of bytes needed to store an object of certain type?
- **Use** `sizeof(object);`
- **Example:**
  ```c
  int a, b[3];
  printf("int:%3d bytes\n", sizeof(int));
  printf("int:%3d bytes\n", sizeof(a));
  printf("array b:%3d bytes\n", sizeof(b));
  ```
Referencing 2D Arrays

- Given `int a[3][5];`
- Equivalent expressions of `a[i][j]`:
  
  *(a[i]+j)*  
  *(*(a+i))[j]*  
  *(((a+i))+j)*  
  *(*(a+i)+j)*  
  *(&a[0][0]+5*i+j)*