What is a Pointer?

• int k; k = 2;
  - k is a symbol for a memory location where 4 bytes of data (the int value 2) are stored.
  - k itself refers to the stored value
  - &k refers to the address of the memory location

• int *p
  - p is a symbol for a memory location where the address of another memory location will be stored
  - the address is where an int value is stored
  - the int value can be accessed using the ‘dereferencing operator’ * as *p
What is a Pointer?

• \( p = \&k; \)

• Hence \( *p = 5 \) is equivalent to \( k = 5 \)
Data Type of a Pointer

- \( *(p+1) = ? \)
  - \( p \) is 0x00; incrementing it by 1 should make it 0x01
  - but, C increments a pointer based on its type
  - since \( p \) is a pointer of type \texttt{int}, +1 results in addition of \texttt{sizeof(int)} [4 bytes here] to \( p \)
  - so, \((p+1)=0x100\) and \( *(p+1)=7 \)

- If you want \( p \) to increment by 1 byte, use typecast: \((\texttt{int}*)((((\texttt{char}*)p)+1))\)
Pointers and Arrays

• \( p = \&\text{arr}[0]; \)

• The symbol \( \text{arr} \) itself is a pointer to the first element of the array

• Hence, \( \text{arr}[i] \) can also be written as \( * (\text{arr} + i) \)
Pointers and Arrays

- int arr[3] = {1, 2, 3}; int *p = arr;
  - p is pointing to arr[0] (=1);

- Step1: printf("%d", *p++)
  - Get value at p: 1 (output)
  - Increment p: p is now pointing to arr[1]

- Step2: printf("%d", *++p)
  - Increment p: p is now pointing to arr[2]
  - Get value at p: 3 (output)

- Step3: printf("%d", +++p)
  - Increment value at p and then print it
    - arr[2]=3+1=4 (output)
Pointers and Strings

• A string in C is simply an array of char values
• Copy the contents of strA into strB:

```c
char strA[] = "Ping";
char strB[] = "Pong";
char *pA = strA;
char *pB = strB;

while (*pA) *pB++ = *pA++;
*pB = '\0';;
```
Pointers and Strings

• Put the reverse of str1 into str2:

```c
char str1[] = "Pointers are fun. Yeah right!";
char str2[30], *p1, *p2;

p1 = str1 + strlen(str1) - 1;
p2 = str2;

while(p1 >= str1) *p2++ = *p1--;
*p2 = '\0';
```
Pointers and Multi-Dimensional Arrays

- `arr[2]` is the pointer to the third row
- So, we can access `arr[2][1]` as `*(arr[2]+1)`
- But `arr[2]` is again same as `*(arr + 2)`
- So, `arr[2][1]` is same as `(*(arr+2)+1)`
- In general, `arr[i][j]` is same as `(*(arr+i)+j)`
Dynamic Memory Allocation

- `int arr[1000];`
  - sets aside $1000 \times \text{sizeof(int)}$ bytes of memory irrespective of whether you use it or not
- Instead, use `malloc()` to allocate memory at runtime depending on requirement
- And then when you are done using it, use `free()` to deallocate it
- `malloc`ed memory will not be automatically freed until process exits
malloc()

• \(\text{int } *p = (\text{int } *)\text{malloc(}\text{sizeof(int)}*N)\)
  - allocates enough memory to hold \(N\) int values
  - returns the starting address of the memory block
  - we store the address in the pointer \(p\)
  - \text{malloc()} returns NULL if memory could not be allocated

• We can use \(*p, *(p+1), ..., *(p+N-1)\) to refer to the integers in the memory block

• But, \(*(p+i)\) is same as \(p[i]\)

• Effectively, we just dynamically allocated an array to hold \(N\) integers
free()

- `free(p)`
  - deallocates the memory addressed by `p`
  - It’s good practice to set the pointer to `NULL`: `p=NULL`

```
int *p

FREE()

p=(int *)malloc(N*sizeof(int))

FREE(p)
```
Dynamic Memory for 2D Arrays

• Recall that each row of a 2D array can be referenced by a pointer to a 1D array

• So for 2D arrays, we need a 1D array of pointers

\[
\text{int **p;}
\]
\[
p = (\text{int **}) \text{malloc}(Nrow \ast \text{sizeof(int *)})
\]

• We just allocated memory to hold \(Nrow\) pointers, accessed as \(*(p+i)\) (or \(p[i]\))

\[
\text{for (i=0; i<Nrow; i++)}
\]
\[
p[i] = (\text{int *})\text{malloc}(Ncol \ast \text{sizeof(int)})
\]

• Each of those pointers now points to a block of memory of size \((Ncol\ast\text{sizeof(int)})\)
Dynamic Memory for 2D Arrays

• To access the integer at row $i$ and column $j$, use
  - $\ast(*\!(p+i)+j) \text{ or } p[i][j]$

• Each $\ast(p+i)$ need not point to a memory block of same size
  - therefore, each column of the array can be of different size

• To free the memory:
  
  for (i=0; i<Nrow; i++) free(p[i]);
  free(**p); p=NULL;

• You may also free only certain columns:
  
# include <stdio.h>
# include <stdlib.h>
int main()
{
    int n, i, *ptr, sum=0;
    printf("Enter number of elements: ");
    scanf("%d",&n);                       // number of elements entered by user
    ptr=(int*)malloc(n*sizeof(int));     // memory allocated using malloc
    if(ptr==NULL) {
        printf("Error! memory not allocated.");
        exit(0); }
    printf("Enter elements of array: ");
    for(i=0;i<n;++i) {
        scanf("%d",ptr+i);              // array values entered by user
        sum+=*(ptr+i); }
    printf("Sum=%d",sum);
    free(ptr);                         // don’t forget free()!
    return 0;
}
Makefile basics

• A makefile is simply a way of associating short names, called targets, with a series of commands to execute when the action is requested
  - Default target: make
  - Alternate target: make clean
Makefile

• Basic macro:  \texttt{CC=gcc}
• Convert a macro to its value in a target:  $$(\text{CC})$$
  - Ex: $$(\text{CC})$$ \texttt{a_source_file.c} gets expanded to \texttt{gcc a_source_file.c}

• Basic makefile:

  \begin{verbatim}
  CC = gcc
  FILES = in_one.c in_two.c
  OUT_EXE = out_executable
  build: $(FILES)
    $(CC) -o $(OUT_EXE) $(FILES)
  \end{verbatim}

• To execute:  \texttt{make build}
Make clean

CC = gcc
FILES = in_one.c in_two.c
OUT_EXE = out_executable
build: $(FILES)
    $(CC) -o $(OUT_EXE) $(FILES)
clean:
    rm -f *.o $(OUT_EXE)

• The target make clean will remove all .o files and the executable
References

- A Tutorial on Pointers And Arrays in C:  
  [h/p://home.netcom.com/~tjensen/ptr/pointers.htm](h/p://home.netcom.com/~tjensen/ptr/pointers.htm)

- Essential C:  

- Reading C Type Declarations:  
  [h/p://unixwiz.net/techtips/reading-cdecl.html](h/p://unixwiz.net/techtips/reading-cdecl.html)

- C Programming Dynamic Memory Allocation:  