Second part

Dr. Alekha Kumar Mishra

String

- Strings in C are simply arrays of characters.
  - Example: `char s [10];`
  - can hold a character string of $\leq$ 9 characters.
  - C does not know where the end of an array is at run time.
  - By convention, C uses a NULL character '\0' to terminate all strings in its library functions
- For example:
  - `char str [10] = {'u', 'n', 'I', 'x', '\0'};`
  - It’s the ‘\0’ (not the size of the array) that determines the length of the string.

String Literals

- String literals/constants are given as a string quoted by double quotes.
  - printf("Long long ago.");
- Initializing string
- char s[10]="unix"; /* s[4] is '\0'; */
- char s[ ]="unix"; /* s has five elements */

Printing with printf()

- char str[ ] = "A message to display";
  printf ("%s\n", str);
- printf knows how much to print out because of the NULL character at the end of all strings.
- When it finds a \0, it knows to stop.
- Example
- char str[]="unix and c";s
- printf("%s", str);
- str[6]='\0';
- printf("\n%s", str);

The C String library

- String functions are provided in an ANSI standard string library.
- Access this through the include file:
  - #include <string.h>
- Includes functions such as:
  - returns length of string
  - Copying strings
  - Concatenating strings

strlen()

- strlen returns the length of a NULL terminated character string:
  - size_t strlen (char * str) ;
- Defined in string.h
- size_t : a type defined in string.h is equivalent to an unsigned int
- The following code has a problem!!
  - char a[5]={‘a’, ’b’, ’c’, ’d’, ’e’};
  - strlen(a);

strcpy()

- Copying a string
  - char * strcpy (char * destination, char * source);
- A copy of source is made at destination
- source should be NULL terminated
- destination should have enough room
  (its length should be at least the size of source)
- The return value also points at the destination.

strcat()

- Concatenation function
  - char * strcat (char * str1, char * str2);
  - Appends a copy of str2 to the end of str1
  - A pointer equal to str1 is returned
- Ensure that str1 has sufficient space for the concatenated string!
  - Array index out of range is be the most popular bug in beginner.

Example

```
#include <string.h>
#include <stdio.h>
int main() {
    char str1[27] = "abc";
    char str2[100];
    printf("%d\n",strlen(str1));
    strcpy(str2,str1);
    puts(str2);
    puts("\n");
    strcat(str2,str1);
    puts(str2);
}
```

Comparing strings

- C strings can be compared for equality or inequality
- If they are equal - they are ASCII identical
- If they are unequal the comparison function will return an int that is interpreted as:
  - < 0 : str1 is less than str2
  - 0 : str1 is equal to str2
  - > 0 : str1 is greater than str2

strcmp()

- int strcmp (char *str1, char *str2) ;
  - Does an ASCII comparison one char at a time until a difference is found between two chars
  - Return value is as stated before
  - If both strings reach a '\0' at the same time, they are considered equal.
- int strncmp (char *str1, char * str2, size_t n);
  - Compares n chars of str1 and str2
  - Continues until n chars are compared or the end of str1or str2 is encountered
- Also have strcasecmp() and strncasecmp() which do the same as above, but ignore case in letters.

strncmp()

```
int main() {
        char str1[] = "The first
        string.";
        char str2[] = "The
        second string.";
        size_t  n, x;
        printf("%d\n",
        strncmp(str1,str2,4) );
        printf("%d\n",
        strncmp(str1,str2,5) );
}
```

sprintf() for string

- int sprintf(char *s, const char *format, ...);
- Instead of printing to the stdin with printf(...), sprintf prints to a string in this case.
- Very useful for
  - formatting a string
  - converting integers or floating point numbers to strings.
- There is also a sscanf for formatted input from a string in the same way scanf works.

Example

```
#include <stdio.h>
#include <string.h>
int main()
{
        char result[100];
        sprintf(result, "%f", (float)17/37 );
        printf(“%s”,result):
        return 0;
 }
```

String to number conversion

- Defined in <stdlib.h> and are often used
- int atoi (char *ptr);
  - Takes a character string and converts it to an integer.
  - White space and + or - are OK.
  - Starts at beginning and continues until something non-convertible is encountered.

Example

String	Value returned
"157"	157
"-1.6"	-1
"+50x"	50
"twelve"	0
"x506"	0

String to number conversion

- long atol (char *ptr) ;
  - Same as atoi except it returns a long.
- double atof (char * str);
  - Handles digits 0-9.
  - A decimal point.
  - An exponent indicator (e or E).
  - If no characters are convertible a 0 is returned.

Example

String	Value returned
"12"	12.000000
"-0.123"	-0.123000
"123E+3"	123000.000000
"123.1e-5"	0.001231

Function

- A self-contained program segment that carries out some specific, well-defined task.
- A function receives zero or more parameters, performs a specific task, and returns zero or one value.
- A function is invoked by its name and parameters.
  - No two functions have the same name in your C program.
- A function is independent:
  - It can be called at any places of your code and can be ported to another program.
- Programs combine user-defined functions with library functions

Function definition format

- Function Prototype:
  - return_type function_name (type$_1$ name$_1$, type$_2$ name$_2$, …, type$_n$ name$_n$);
- Function Definition:
  - return_type function_name ((type$_1$ name$_1$, type$_2$ name$_2$, …, type$_n$ name$_n$)

    {

    ….statements…

    }
- Function-name: any valid identifier
- Return-value-type: data type of the result (default **int**)

Function definition format

- **void** - indicates that the function returns nothing
- Parameter-list: comma separated list, declares parameters
- Variables can be declared inside the block
- Functions can not be defined inside another function
- Returning control
  - If nothing returned → return; (optional)
  - If something returned → return expr;

Function prototype

- Function name
- Parameters – what the function takes in
- Return type – data type function returns (default **`int`**)
- Used to validate functions
- Prototype only needed if function definition comes after use in program
- The function with the prototype
- `int maximum( int, int, int );`
  - Takes in 3 **`int`**s
  - Returns an **`int`**

Example

```
#include <stdiio.h>
int maximum(int , int , int);

int main(){
    int a,b,c,m;
    printf("enter three integers");
    scanf(" %d %d %d ",&a,&b,&c);
    m=maximum(a,b,c);
   printf(" maximum is %d",m);

  return 0;
}

int  maximum(int x,int y,int z){
    int max = x;
    if(y > max)
        max = y;
   if(z > max)
        max = z;
   printf("in function max =
    %d",max);
    return max;
}
```

→ Each argument can be a valid c expression that has a value

→For example:

x = func1(x+1,func1(2,3,4),5);

→Parameters x y z are initialized by the value of a b c

→Type conversions may occur if types do not match

Call by Value and Call by Reference

- Used when invoking functions
- Call by value
  - Copy of argument passed to function
  - Changes in function do not effect original
  - Use when function does not need to modify argument
  - Avoid accidental changes
- Call by reference
  - Passes original argument
  - Changes in function effect original
  - Only used with trusted functions

Example Call by Value

```
#include<stdio.h>
int twice(int x)
{
    x=x+x;
    return x;
}
int main()
{
    int x=10,y;
    y=twice(x);
    printf("%d,%d\n",x,y);
}
```

Call by Reference

- Address of the parameters are passed instead of its value
- So, you are accessing the original variable of calling function not a copy of it
- More detail in pointer section

```
#include<stdio.h>
void swap(int*, int*);
int main(){
    int x=10,y=30;
    swap(&x,&y);
    printf("%d,%d\n",x,y);
}
void swap(int *x,int *y){
    int temp;
    temp=x;
    x=y;
    y=temp;
    return;
}
```

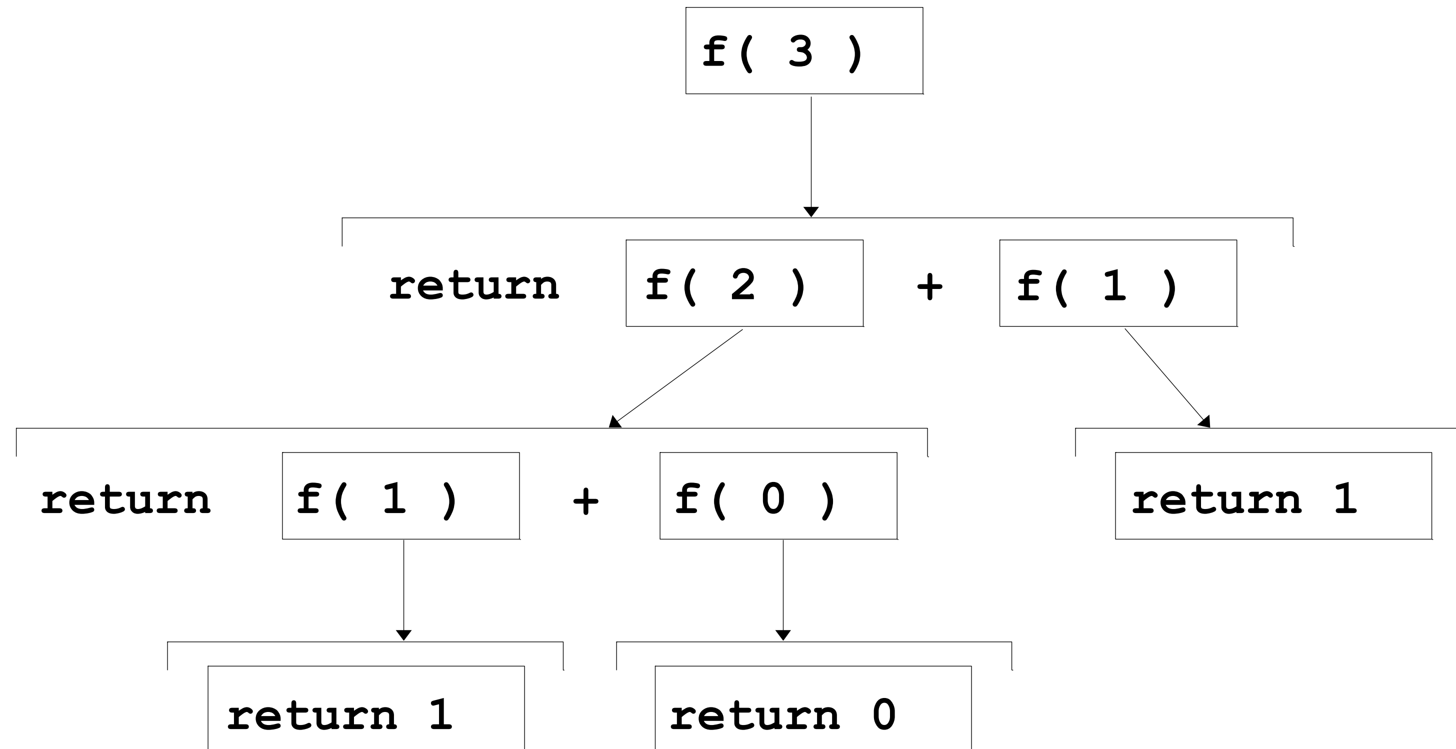
- Functions that call themselves
- Can only solve a base case
- Divides a problem up into
  - What it can do
  - What it cannot do
- What it cannot do resembles original problem
- The function launches a new copy of itself (recursion step) to solve what it cannot do
- Eventually base case gets solved

Example

- Fibonacci series: 0, 1, 1, 2, 3, 5, 8...
- Each number is the sum of the previous two
- Can be solved recursively:
- fib(n) = fib(n - 1) + fib(n – 2)

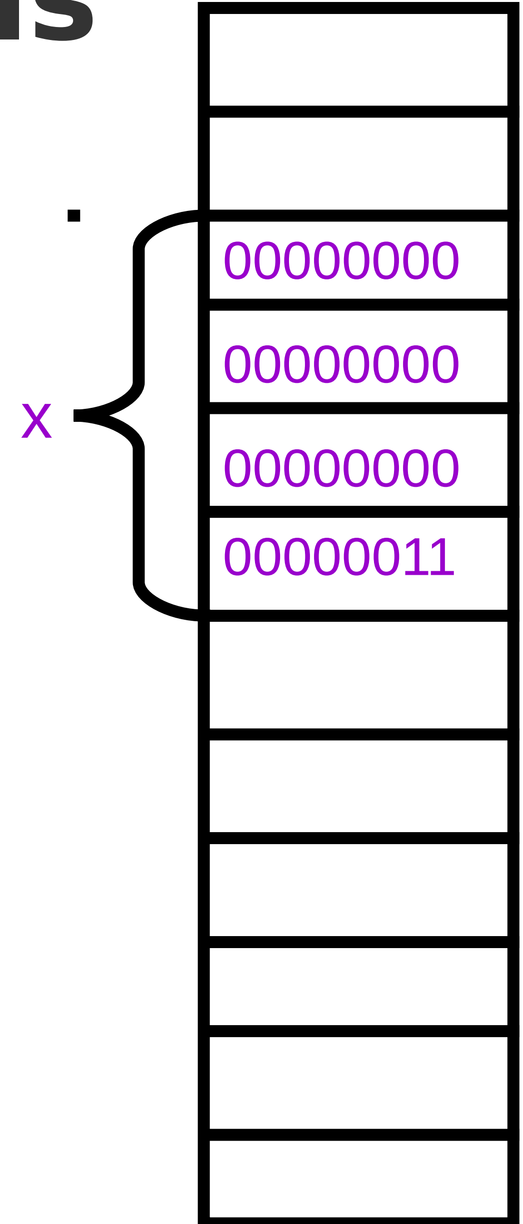
```
long fibonacci( long n )
{
   if (n == 0 || n == 1)  // base case
    return n;
   else
    return fibonacci( n - 1) + fibonacci( n
– 2 );
}
```

Recursion call steps



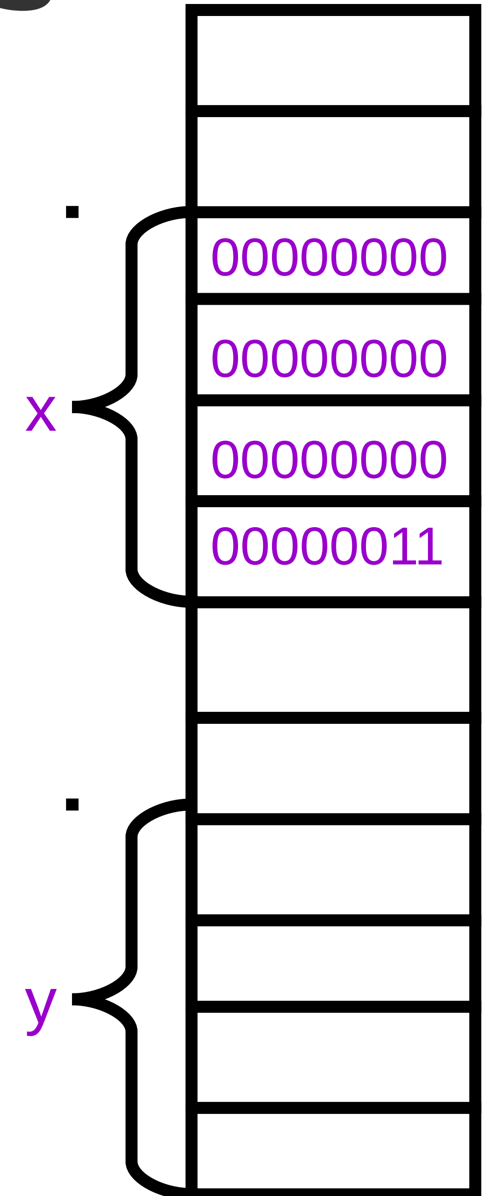
Pointer Fundamentals

- When a variable is defined the compiler (linker/loader actually) allocates a real memory address for the variable.
- int x; will allocate 4 bytes in the main memory, which will be used to store an integer value.
- When a value is assigned to a variable, the value is actually placed to the memory that was allocated.
- x=3; will store integer 3 in the 4 bytes of memory.



Pointer Fundamentals

- When the value of a variable is used, the contents in the memory are used.
- y = x; will read the contents in the 4 bytes of memory, and then assign it to variable y.
- &x can get the address of x. (referencing operator &)
- The address can be passed to a function:
- scanf("%d", &x);
- The address can also be stored in a variable called pointers



- Can be used to access data stored in memory
- Use of pointers gives the power and flexibility.
- C uses pointers explicitly with
  - Arrays,
  - Structures,
  - Functions.
- Provides a way to return multiple elements from a function via function arguments
- It produces compact and efficient code

Pointers

- We can have a pointer to any variable type.
- The operator **&** gives the address of a variable.
- The ***indirection*** or dereference operator ***** gives the contents of an object ***pointed to*** by a pointer

Pointer Declaration

- To declare a pointer variable
- type * pointername;
- For example:
  - int * p1;   p1 is a variable that tends to point to an integer, (or p1 is a int pointer)
  - char *p2;
  - p1 = &x;   /* Store the address in p1 */
  - scanf("%d", p1); /* i.e. scanf("%d",&x); */
  - p2 = &x;   /* Will get warning message */

Pointer intialization

```
int quantity;
int *p; //declaration
p=&quantity; //initialization

int x, *p=&x; // all steps in one

// Assignment of NULL or 0 (zero)
   int *p=NULL;
   int *p=0;

// this is invalid, absolute address
    int *p=5360;
```

Example

```
int main(){
    int m=3, n=100, *p;
    p=&m;
    printf("m is %d\n",*p);
    m++;
    printf("now m is %d\n",*p);
    p=&n;
    printf("n is %d\n",*p);
    *p = 500;
    printf("now n is %d\n", n);
   return 0;
}
```

Pointer to a pointer

- pointer variable p2 contains address of p1
- Also known as *multiple indirections*

```
main()
{
   int x, *p1, **p2;
   x=100;
   p1=&x;
   p2=&p1;
   printf("%d", **p2);
}
```

Pointers for call by reference

```
void min_max(int a, int b,
        int *min, int *max){
    if(a>b){
        *max=a;
        *min=b;
    }
   else{
        *max=b;
        *min=a;
    }
}
```

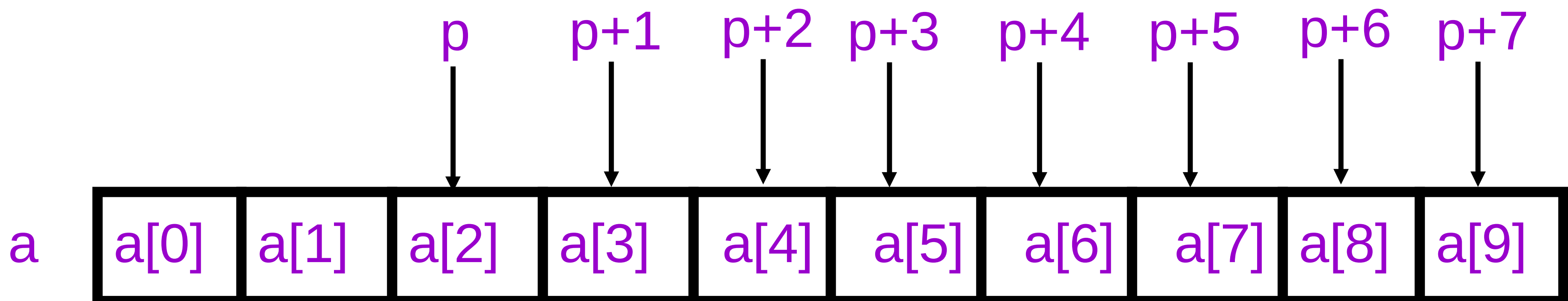
```
int main()
{
    int x,y;
    int small,big;
    printf("Two integers: ");
    scanf("%d %d", &x, &y);
    min_max(x,y,&small,&big);
    printf("%d <= %d", small, big);
    return 0;
}
```

Pointer Arithmetic

When pointer variables points to an array element, there is a notion of adding or subtracting an integer from to/from pointer

```
int a[ 10 ], *p;
p = &a[2];
*p = 10;
*(p+1) = 10;
printf("%d", *(p+3));
```

```
int a[ 10 ], *p;
a[2] = 10;
a[3] = 10;
printf("%d", a[5]);
```



Pointer Arithmetic : More examples

```
int a[10], *p, *q;
    p = &a[2];
    q = p + 3;          /* q points to a[5] now */
    p = q – 1;          /*  p points to a[4] now */
    p++;                /*  p points to a[5] now */
    p--;                      /* p points to a[4] now */
    *p = 123;           /*  a[4] = 123 */
    *q = *p;            /*   a[5] = a[4]   */
    q = p;              /*   q points to a[4] now */
    scanf("%d", q)    /* scanf("%d", &a[4])  */
```

Pointer Arithmetic

If two pointers point to elements of a same array, then there are notions of subtraction and comparisons between the two pointers.

```
int a[10], *p, *q , i;
p = &a[2];
q = &a[5];
i = q - p;        /* i is 3*/
i = p - q;        /* i is -3 */
a[2] = a[5] = 0;
i = *p - *q;      /* i = a[2] – a[5]  */
p < q;            /* true */
p == q;           /* false */
p != q;           /* true */
```

Pointer and Arrays

- The value of an array name is also an address.
- In fact, pointers and array names can be used interchangeably in many (but not all) cases.
- E.g. int n, *p;   p=&n;
-         n=1; *p = 1; p[0] = 1;
- The major differences:
  - Array names come with valid memory spaces where they "point" to and you cannot "point" their names to other places.
  - Pointers do not point to valid memory space when they are created. You have to point them to some valid space (initialization)

Example

```
int main(){
   static int x[5]={10, 11, 12, 13, 14};
   int i;
   for(i=0;i<5;i++){
      printf(“x[i] = %d  *(x+i)=%d”,x[i],*(x+i));
      printf(“&x[i] = %p (x+i)=%p”,&x[i],(x+i));
  }
}
```

Pointer and Arrays

- Array name is like a constant pointer which points to the first element of the array.

```
int a[10], *p, *q;
p = a;          /* p = &a[0] */
q = a + 3;      /* q = &a[0] + 3 */
a ++;           /*  illegal !!! */
```

- Therefore, we can "pass an array" to a function. Actually, the address of the first element is passed.

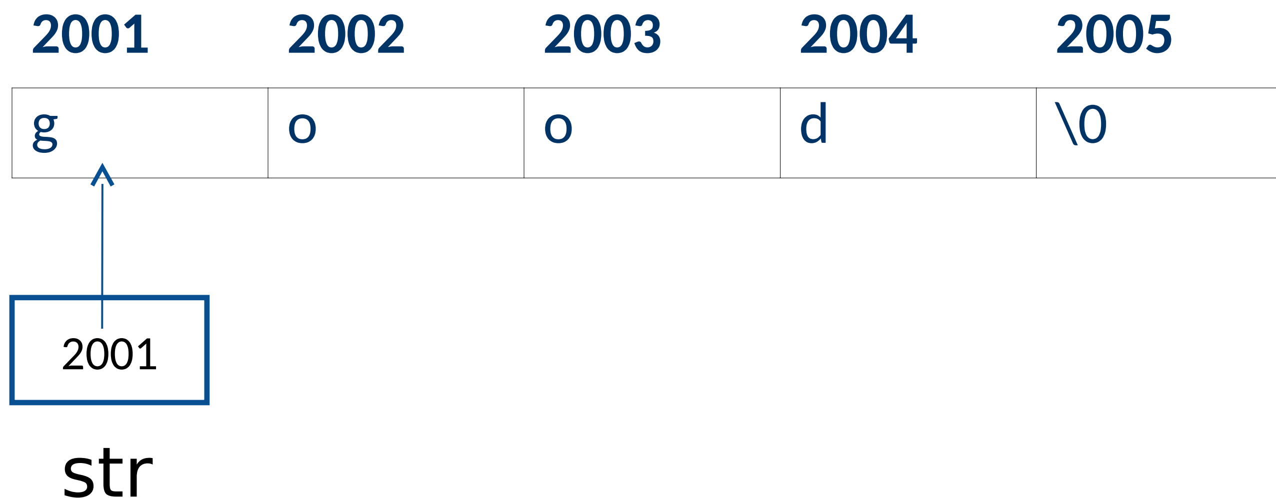
```
int a[ ] = { 5, 7, 8 , 2, 3 };
sum( a, 5 );  /* Equal to sum(&a[0],5)  */
```

Pointers and 2D Arrays

p	pointer to first row
p+i	pointer to ith row
*(p+i)	pointer to first element of ith row
*(p+i)+j	pointer to jth element in ith row
((p+i)+j)	value stored in cell (i,j) (ith row and jth column)

Pointer and string

- char str[5]=“good”;
- char *str=“good”;



Function returning pointer

```
main()
{
    int a=10;
    int b=20;
    int *p;
    p=larger(&a,&b);
    printf("%d",*p);
}
int * larger(int *x,int *y)
{
    if(*x>*y)
        return x;
    else
        return y;
}
```

Pointer to a function

- A pointer to function is declared as
  - type (*fptr) ();
  - This tells compiler that fptr is pointer to function and returns type value
- Example
  - double mul(int, int);
  - double (*p1)();
  - p1=mul;
  - point p1 to mul
  - we can now use p1 to call function mul
    - (*p1)(x,y);

Allocating Memory for a Pointer

The following program is wrong!

```
#include <stdio.h>
int main()
{
    int *p;
    scanf("%d",p);
    return 0;
}
```

This one is correct:

```
#include <stdio.h>
int main()
{
    int *p;
    int a;
    p = &a;
    scanf("%d",p);
    return 0;
}
```

Allocating Memory for a Pointer

There is another way to allocate memory so the pointer can point to something:

```
#include <stdio.h>
#include <stdlib.h>
int main(){
    int *p;
    p = (int *) malloc( sizeof(int) );   /* Allocate 4 bytes */
    scanf("%d", p);
    printf("%d", *p);
    free(p);          /* This returns the memory to the system.*/
                      /* Important !!! */
}
```

Allocating Memory for a Pointer

- Prototypes of malloc() and free() are defined in stdlib.h
  - void * malloc(size_t number_of_bytes);
  - void free(void * p);
- You can use malloc and free to dynamically allocate and release the memory;
  - int *p;
  - p = (int *) malloc(1000 * sizeof(int));
  - for(i=0; i<1000; i++)
  -     p[i] = i;
  - p[1000]=3;   /* Wrong! */
  - free(p);
  - p[0]=5;   /* Wrong! */

End of part 2