PROBLEM SOLVING USING C

UNIT-2

TEXT PROCESSING & STRING MANIPULATION STRUCTURES

Vibha Masti

Feedback/corrections: vibha@pesu.pes.edu

Functions

1) Function declaration

names not req.

```
rettype fun( t1 , t2);
```

(also called prototype / signature)

2) Function definition

```
rettype func(t1 p1 , t2 p2) {

        S1;
        S2;
        :

        return ret
}
```

Usage

```
int main() {
        :
    func();
        :

}
```

functions can be invoked / called inside any other function (main also)

functions with return values are stored in variables

1) void functions, no parameters - simplest func

```
#include <stdio.h>

void foo();          // prototype / declaration

int main() {
    .
    .
    foo();
    return 0;
}

void foo() {         // definition
    printf("Sup\n");
}
```

indicates to compiler that function exists (name valid)

non-executable statement

actual function code

2) return type, parameter list

```
returntype <funcname> (parameter list) {
    body
}
```

- no explicit global variable
- :: ⟶ scope resolution operator
- parameter list should match function declaration

Variable Scope

```
#include<stdio.h>
int a = 1; ⟶ global
void foo() {
    int a = 2; ⟶ local to foo
    :
    printf("%d\n", a); ⟶ 2
}
int main () {
    printf("%d\n", a); ⟶ 1
    :
    return 0;
}
```

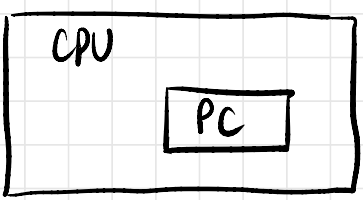
Return Datatype

- store local values in higher scope by reassigning
- copying while transferring variables between function
- program counter — keeps track of execution

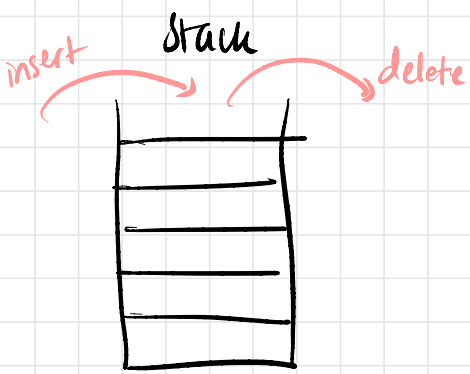
```
int main() {
100     sum(a,b);
        ;
}

2000 int sum(int x, int y) {
        ;
    return x+y;
}
```

transfer of control (program counter)

- execution of main paused, execution of sum started



- first PC → 100
- goes to 2000, loses 100
- stack has 100 left

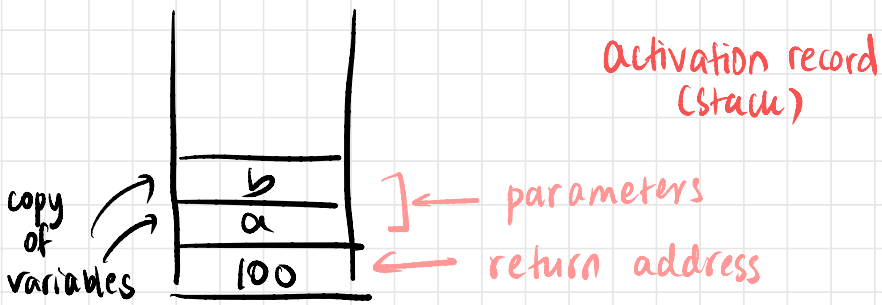


when execution of main paused, its location pushed to stack

current contents of PC pushed to stack

Execution of function (pass by value)

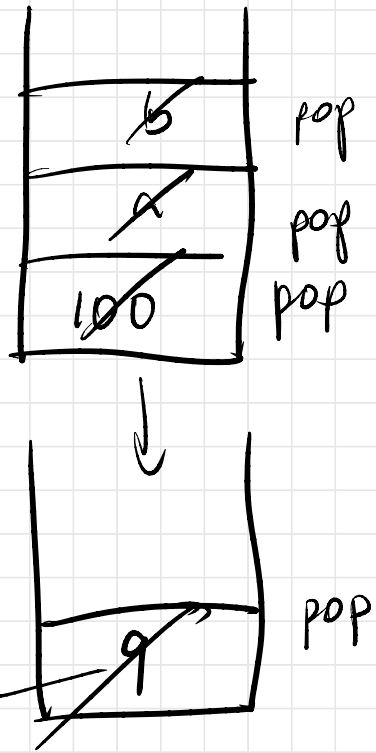
```
100    sum(a,b);    // in main function

2000   int sum(int x, int y) {

       }
```



- a & b popped and copied to x and y locations
- PC changes to destination address 2000
- at return statement, 100 popped and put into PC
- execution resumes from call address
- any change in x & y does not reflect in a & b

Returning values

```
int s;
s = sum(a,b);
  .
  .
  .
int sum(int x, int y) {
  .
  .
  .
    return c;
}
```



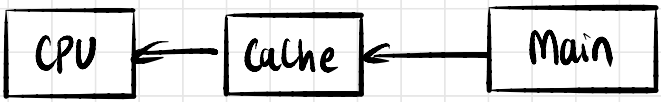
Preserving Environment

- Stack
- Transfer of control from calling program to called program
- Only one program counter

local variables on stack

allocated at runtime (stack stores loc)

dynamic memory




```
int sum(int x, int y) {
    int c;
    c = x+y;
    printf("\n%d \t %d\n", c, c);
    return c;
}
```

O/P

9 -1. #QNAND

↑ user value from cache

homework

Recursion

function calling itself

Fibonacci

$$f(n) = f(n-1) + f(n-2)$$

Multiple files

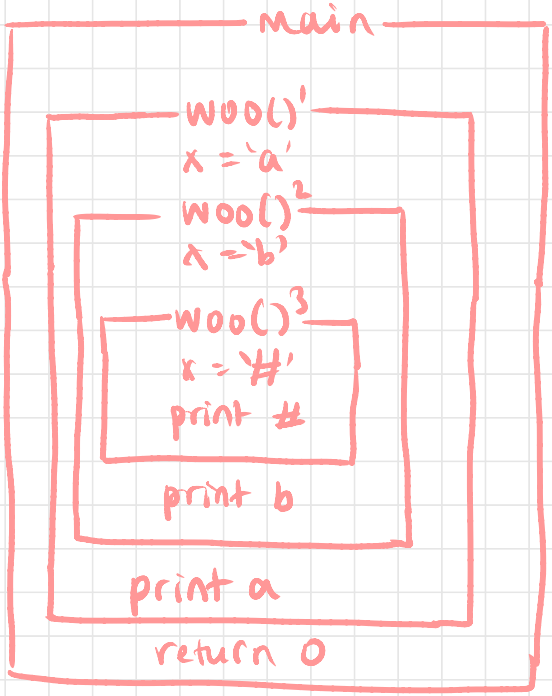
using interfaces

- one file: function declarations, globals
- one file: function definitions
- main program: includes all files

Recursion

```
void woo();
int main() {
    woo();
    ;
}

void woo() {
    char x;
    if ((x = getchar()) != '#')
        woo();
    printf("%c",x);
}
```

Execution



reverses a string by reading char by char.

Q: Check if no. is palindrome or not

main - driver program

can also use " ", but convention is to use <> for library headers and " " for user-defined

```
#include <stdio.h>
#include "is_palin.h"
#include "is_palin.c"
int main() {
    int num;
    printf("Enter a number: ");
    scanf("%d", &num);

    if (is_palin(num)) {
        printf("%d is a palindrome\n", num);
    }
    else {
        printf("%d is not a palindrome\n", num);
    }
}
```

HOMEWORK: try

.c for func. definitions

is_palin.c (same folder / path)

```
#include "is_palin.h"

int is_palin(int n) {
    int copy = n, rev = 0;

    while (copy) {
        rev *= 10;
        rev += copy % 10;
        copy /= 10;
    }

    return (rev == n);
}
```

is_palin.h ⟶ only function headers

```
int is_palin(int);
```

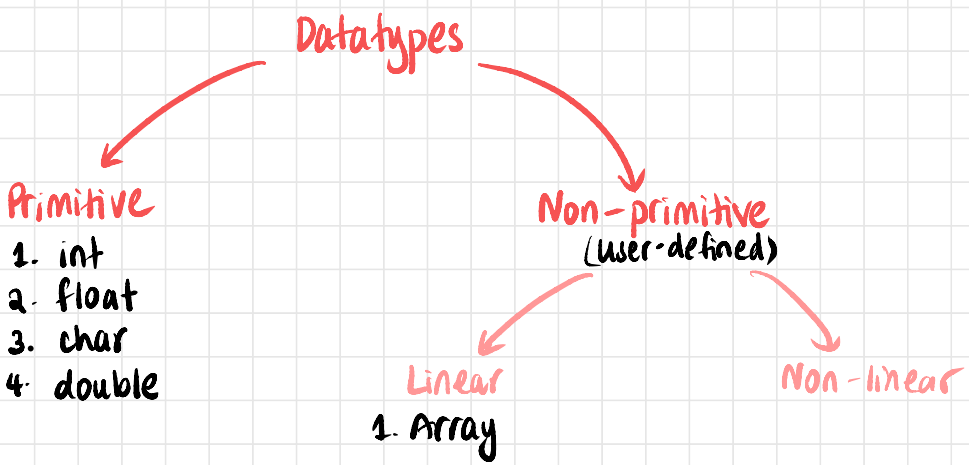
Execution

- speed of execution depends on runtime, not compile time
- compile time increases, but debugging and readability improved

- auto, static, extern, register — storage classes

Explicitly compile multiple files — try this HOMEWORK

```
gcc f1.c -o f1 f2.c -o f2
```



Array

1. Homogeneous data
2. Consecutive memory locations
3. Same name
4. Accessed together

int A[10];

Reading an array

```
for (int i=0; i<n; ++i) {
    scanf("%d", &A[i]);
}
```

Accessing elements - indexing

	0	1	2	3	4	5	6	7
A =	3	2	7	9	5	2	4	3

← indicies

numbering starts from 0

A[0] → 3
A[6] → 4

Q: Search an array for an element

```
// Method 1
int main() {
    int arr[30], n, key, loc = -1;

    printf("Enter no. of elements: ");
    scanf("%d", &n);

    printf("Enter the elements:\n");
    for(int i=0; i<n; ++i)
        scanf("%d", &arr[i]);

    printf("Enter element to search for: ");
    scanf("%d", &key);

    for(int i=0; i<n; ++i) {
        if (arr[i] == key) {
            loc = i;
            break;
        }
    }

    if (loc+1) printf("%d found at index %d\n", key, loc);
    else printf("%d not found in array\n", key);

    return 0;

}
```

cannot change array length (static memory)

Note: C has no way of checking if an array goes out of bounds (No exception handling)

Using global arrays

```
// Method 2 - global arrays

int ARR[30];            // global variable - can be accessed everywhere

int search(int);        // function to search global array

int main() {
    int n, key, loc = -1;

    printf("Enter no. of elements: ");
    scanf("%d", &n);

    printf("Enter the elements:\n");
    for(int i=0; i<n; ++i)
        scanf("%d", &ARR[i]);

    printf("Enter element to search for: ");
    scanf("%d", &key);

    loc = search(key);

    if(loc+1) printf("%d found at index %d\n", key, loc);
    else printf("%d not found in array\n", key);

    return 0;
}

int search(int key) {    // uses global array ARR
    for(int i=0; i<n; ++i) {
        if (ARR[i] == key) return i;
    }
    return -1;
}
```

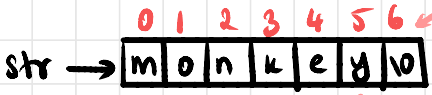
Passing arrays — pass starting address of array
array name is pointer to first element

```
int search(int [], int, int); // int [] is int array

int main() {
    int arr[30], n, key, loc = -1;

    printf("Enter no. of elements: ");
    scanf("%d", &n);

    printf("Enter the elements:\n");
    for(int i=0; i<n; ++i)
        scanf("%d", &arr[i]);

    printf("Enter element to search for: ");
    scanf("%d", &key);

    loc = search(arr, n, key);

    if(loc+1) printf("%d found at index %d\n", key, loc);
    else printf("%d not found in array\n", key);

    return 0;
}

int search(int arr[], int n, int key)
    for(int i=0; i<n; ++i) {
        if(arr[i] == key) return i;
    }
    return -1;
}
```

passing array to function

Working with strings

- C only has char arrays, not strings



added to the end of every string at compile time

null character → returns value 0

Looping through a string

```
int main() {

  char str[] = "monkey";

  for (int i=0; str[i] != '\0'; ++i) {
      ...
  }

            OR

  for (int i=0; str[i]; ++i) {
      ...
  }

            OR

  for (int i=0; i<strlen(str); ++i) {
      ...
  }
```

→ in string.h header file

Finding length of string

does not need size if initialised on same line

```
char str[] = "Hello";
int len = 0;
for (len = 0; str[len]; ++len);
```

declared outside loop for scope

empty loop

Pre-declaring a string/array

need to specify size for compiler (static mem)

```
char A[30];
scanf("%s", A);
```

max size

address of starting location stored in A

ends string with '\0' by default

Manually reading character by character

```
char ch, str[30]; int i = 0;
while ((ch = getchar()) != '\n') {
    str[i++] = ch;
}
str[i] = '\0';
```

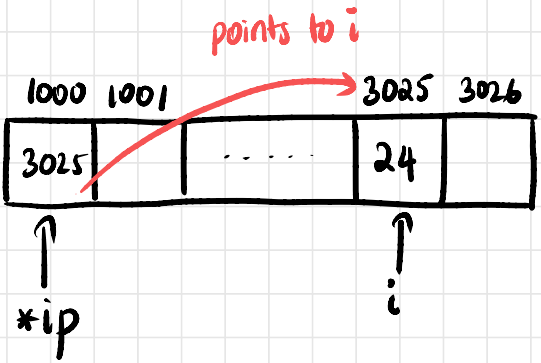
Pointers

- variables that store addresses

```
int i;
int *ip;
ip = &i;
```

`*` indicates pointer of type

`&` gets address of variable



ip stores the location of i

it points to i

- While declaring a pointer, * is used
- Pointers are of type int*, char*, float* etc

While declaring multiple pointers

```
int *p1, *p2, *p3;   ✓
int* p1, p2, p3;     ✗
```

To find the value stored in the variable that the pointer points to,

```
int i = 30;
int *pi = &i;
```

→ pointer of type int* can only store addresses of ints

```
printf("i : %d\n", i);
```

→ 30

```
printf("&i: %d\n", &i);
printf("pi: %d\n", pi);
```

→ address of i (throws warning)

```
printf ("*pi: %d\n", *pi);
```

→ 30

While dereferencing a pointer, * is used.

Only while declaring a pointer does the preceeding * differentiate it from regular variables. It does not act as a dereferencer here.

All following uses of *pi dereferences the pointer and returns the r-value of the variable it points to.

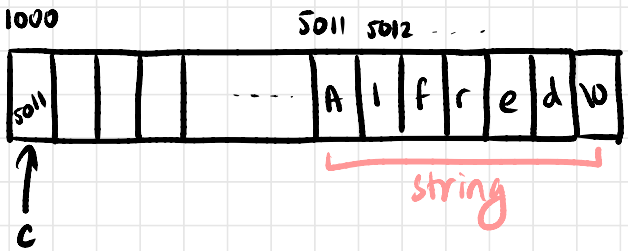
Arrays are essentially pointers.

Character Arrays as Pointers

```
char *c;
c = "Alfred";
```

```
printf("%s \t %p", c, c);
```

→ Alfred → address 321324 (hex)



c: starting address of string (5011)
*c: what is stored at starting address (A)

```
int i = 321;
int *ip;
ip = &i;
```

```
char *cp;
```

→ throws warning

```
cp = ip;
```

→ incompatible pointer types

```
printf("%d \t %p \n", i, &i);
printf("%d \t %p \n", *ip, ip);
printf("%d \t %p \n", *cp, cp);
```

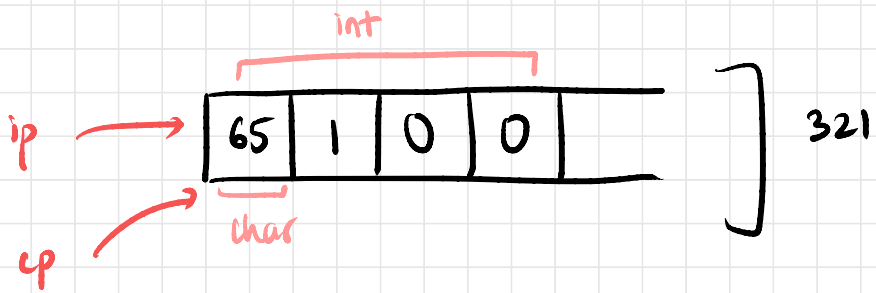
	(hex)
→ 321	662251
→ 321	662251
→ 65	66 2251

Binary 321 → 1 0100 0001 → char is 8-bit

excess bit

65

Note:
cp & ip point to same locations but *cp is only 1 byte while *ip is 4 bytes.



Incrementing Pointers / Pointer Augmentation

Pointers increment by the size of their types

- char ptrs increment by 1 byte
- int ptrs increment by 4 bytes

```
int i = 546;
int *ip;
ip = &i;
```

→ | 34 | 02 | 00 | 00 |

```
char *cp;
cp = ip;
```

```
printf("%d \t %p \n", i, &i);
printf("%d \t %p \n", *ip, ip);
```

(hex)

→ 546 662251
→ 546 662251

```
printf("%d \t %d \n", *(cp), *(++cp));
```

→ 2 2

printf statements are right to left

cp is incremented first, then displayed twice

order of operations

```
int i[] = {1, 5, 7, 9}
int *ip = i;

printf("%d\n", *++ip);          ⟶ 5

ip = i;

printf("%d\n", *ip++);          ⟶ 1
printf("%d\n", *ip);            ⟶ 5

ip = i;
printf("%d   %d\n", *ip++, *ip) ⟶ 1  1
```

execution

```
ip = i;

printf("%d\n", ++*ip);          ⟶ 2

ip = i;

printf("%d\n", ++(*ip));        ⟶ 2

ip = i;

printf("%d\n", (*ip)++);        ⟶ 1
printf("%d\n", *ip);            ⟶ 2
```

*ip++ ⟶ *(ip++)

incrementation > dereferencing

Swap Function Using Pointers

```
void swap (int*, int*);

int main() {
    int a = 4;
    int b = 3;

    printf("a: %d, b: %d\n", a, b);

    swap(&a, &b);

    printf("a: %d, b: %d\n", a, b);

    return 0;
}
```

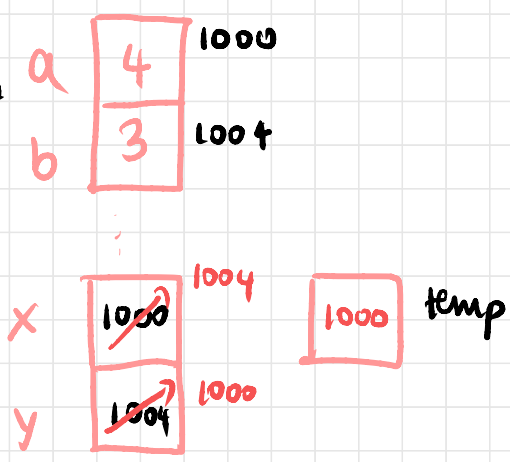
→ should affect values of a & b

Version 1

```
void swap(int *x, int *y) {

    int *temp;
    temp = x;
    x = y;
    y = temp;

}
```

DOES NOT WORK

only changes x & y in function

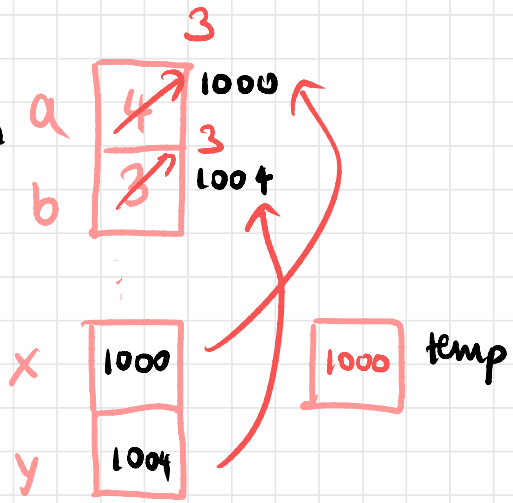


Version 2

```
void swap(int *x, int *y) {

    int *temp;
    temp = x;
    *x = *y;
    *y = *temp;

}
```

DOES NOT WORK

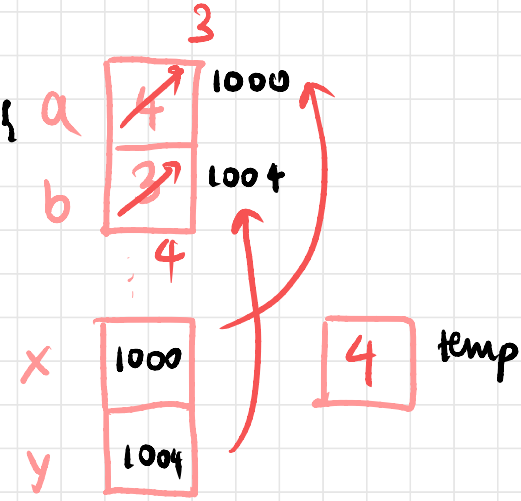
loses value of a



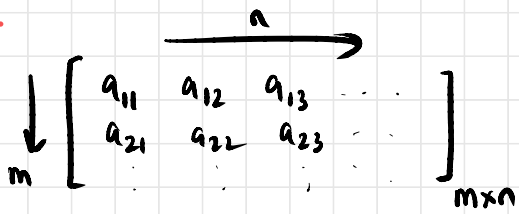
Version 3

```
void swap(int *x, int *y) {

    int temp;
    temp = *x;
    *x = *y;
    *y = temp;

}
```

WORKS



Matrix



1) Row-major

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}$$

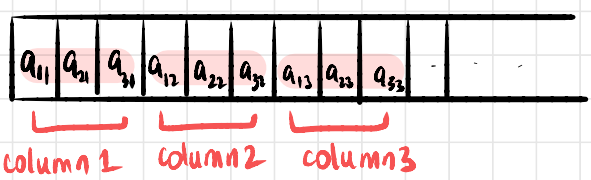
most compilers use row-major order

Storage:

2) Column-Major

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}$$

Storage:



Declaring, Reading and Indexing a Matrix

```
int A[10][10];
```

columns (max)

rows

```
int m,n;

printf("Enter order(m & n): ");
scanf("%d %d", &m, &n);

printf("Enter elements:\n");

//reading elements
for(int i=0; i<m; ++i) {
    for(int j=0; j<n; ++j) {
        scanf("%d", &A[i][j]);
    }
}
```

Storage classes

- auto
- register
- extern
- static

auto, register, extern: automatic storage class; within functions

register: frequent access; fast registers

static: static storage class

auto

- default storage class (even without auto specifier)
- variables within a block; local to block
- all variables we have used so far.

register

- auto storage class but stored in faster registers
- cannot use & (address of) operator on such variables → throws error

extern

- identifiers in included files; functions default extern storage class (global variables)

external.c

```
int x = 400;
```

main.c

```
extern int x;
printf("%d\n", x);
```

→ 400

Static

still only local scope

- preserve value even outside of scope
- hold on to value of their last use in their scope.

main.c

```
void sup() {

    static int x = 1;
    printf("Function call #%d\n", x++);

}

int main() {
    for (int i=0; i<5; ++i) {
        sup();
    }
    // printf("%d\n", x); ⟶ error; scope still maintained
    return 0;
}
```

OUTPUT

```
Function call #1
Function call #2
Function call #3
Function call #4
Function call #5
```

regular (auto) variables do not retain values after leaving scope

static: no memory re-allocated