Programming & Data Structure Laboratory

Arrays, pointers and recursion

Day 5, August 5, 2014

- Pointers and Multidimensional Array
- Function and Recursion

Counting function calls in Fibonacci

```
#include<stdio.h>
int total_call=0; /* Declare global variable*/
int Fib(int n){
total_call++;  if(n==0)  return 0;   if(n==1)  return 1;
return(Fib(n-1)+Fib(n-2));
}
int main(void){
int n, fib_val;
printf("\n n = "); scanf("%d",&n);
fib_val=Fib(n);
printf("\n value = %d and no. of recursive calls=%d\n",
    fib_val,total_call);}
```

Pointers

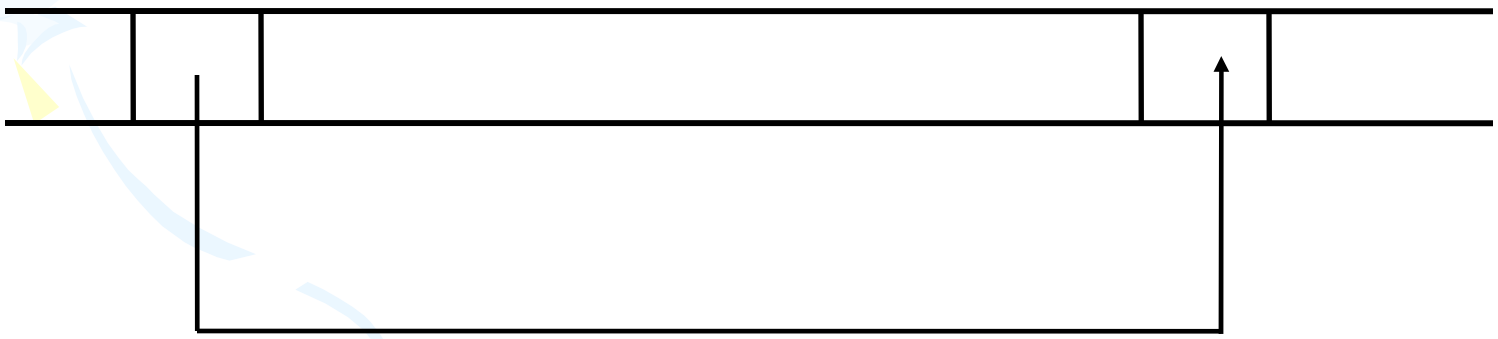
int *p

int x

p = &x

int *p

int x



What is *p? If x = 5, what is *p = ?

An example - swap

```
#include<stdio.h>
void swap(int* x, int* y)
{  int temp;
   temp=*x;*x=*y;*y=temp;
   return;
}

int main(void) {
int a=5,b=4;
swap(&a,&b);
printf("\na=%d, b=%d \n",a,b);
return 0;
}
```

Pointers and sizeof operator

```
#include<stdio.h>


int main(void)
{
  printf("\n The size of pointer to char = %d \n", sizeof(char *),);
  printf("\n The size of pointer to uchar = %d \n",
    sizeof(unsigned char *));
  printf("\n The size of pointer to int = %d \n", sizeof(int *));
  printf("\n The size of pointer to long double = %d \n",
    sizeof(long double *));
  return 0;
}
```

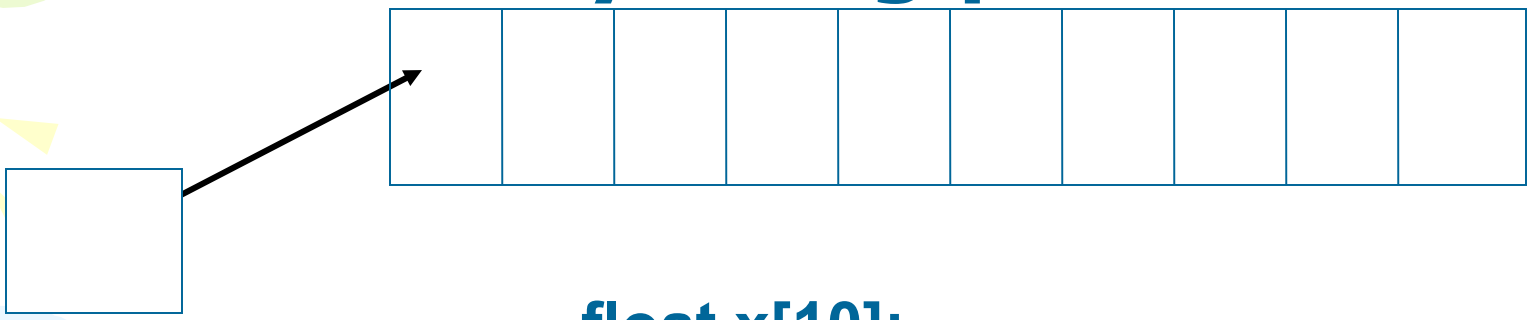
- Write a program to count the number of calls in the recursive version of Fibonacci number computation without using global variables. Use pointers and pass it to functions.

Fib. recursive without global variable

```
#include<stdio.h>
int Fib(int n, int* total_call)  {
(*total_call)++;
if(n==0)  return 0;   if(n==1)  return 1;
return (Fib(n-1,total_call)+Fib(n-2,total_call));
}

int main(void)  {
int n, fib_val, total_call=0;
printf("\n n = "); scanf("%d",&n);
fib_val=Fib(n,&total_call);
printf("\n value = %d and no. of recursive calls=%d\n",
    fib_val,total_call);}
```

1D array using pointers



float *p;

float x[10];

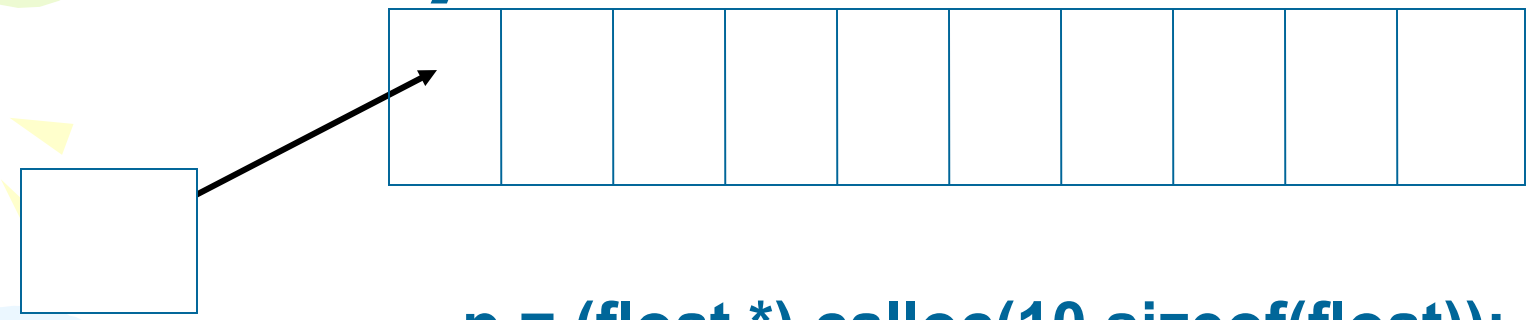
p = &x[0];

Show the pointers correctly for the following statements:

p = &x[6];

p = p+2;

Dynamic allocation



float *p;

p = (float *) calloc(10,sizeof(float));

Show the pointers correctly for the following statements:

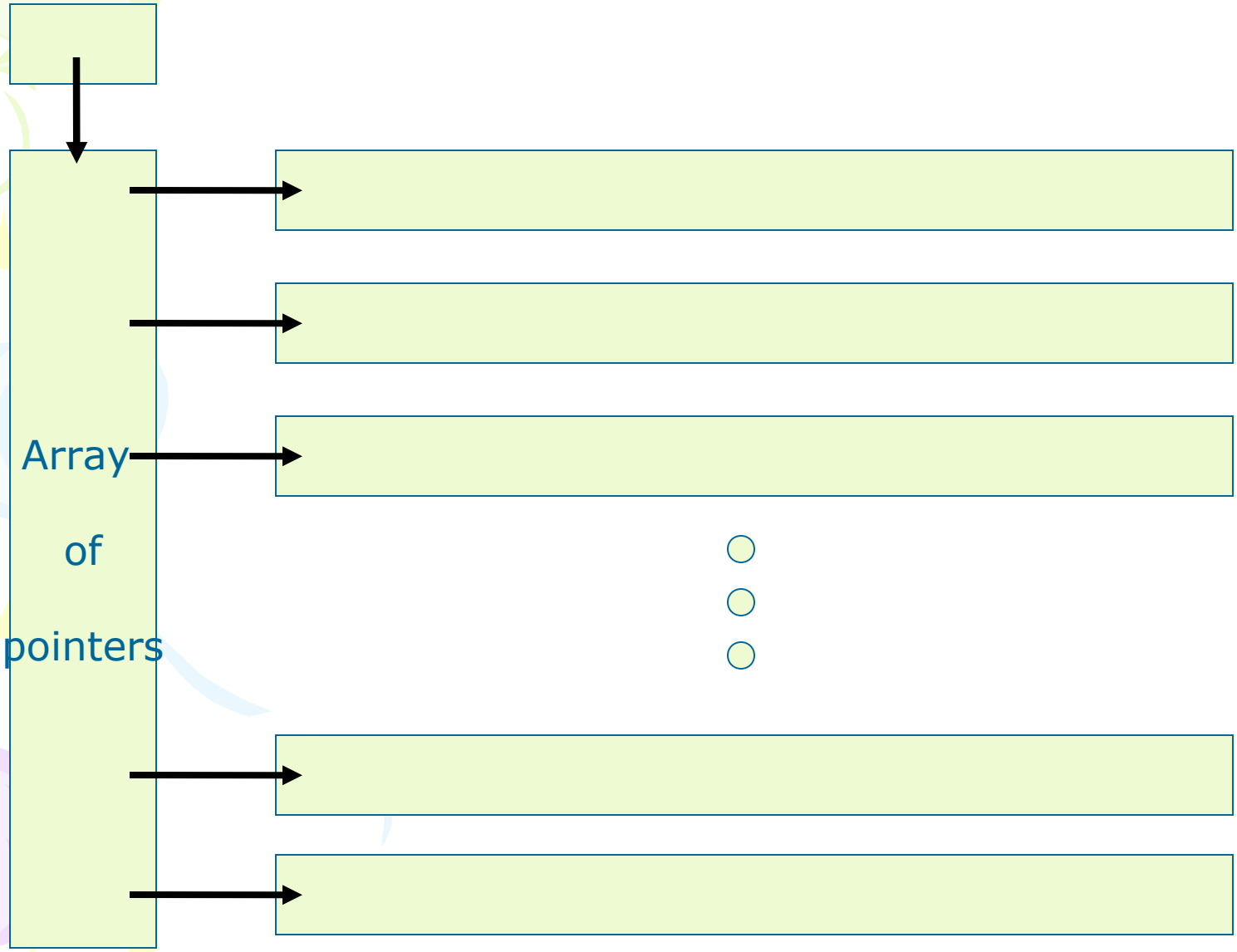
p = &p[6];

p = p+2;

```
#include<stdio.h>
#include<stdlib.h>
int* allocate1D(int row){
int *S;
S=(int *)calloc(row,sizeof(int));
if(S==NULL) { printf("\n No space \n"); exit(0);  }
return S;
}

int main(void){
int *S1,row;
printf("\n How many rows? "); scanf("%d", &row);
S1=allocate1D(row);
return 0;
}
```

Dynamic allocation of 2D array



- Write a small piece of code to allocate a two dimensional matrix using pointer to pointer.

```
#include<stdio.h>
#include<stdlib.h>
int** allocate2D(int row,int col){  int **S,k;
S=(int **)calloc(row,sizeof(int *));
if(S==NULL)  {  printf("\n No space \n"); exit(0);  }
for(k=0; k<row; k++)  {
S[k]=(int *)calloc(col,sizeof(int));
if(S[k]==NULL)   { printf("\n No space \n"); exit(0);  }
}
return S;
}
int main(void){  int **S2,row,col;
printf("\n How many rows? "); scanf("%d",&row);
printf("\n How many columns? "); scanf("%d",&col);
S2=allocate2D(row,col);
return 0;
}
```

- Write a small piece of code for multiplying two two-dimensional matrices using pointer to pointer and passing them as arguments to a function.

Functions

- Functions

Functions

- Given a complicated job, it is always better to break it up into small pieces and solve them. This is the main idea of functions.
- A **function** carries out a specific function depending on the parameters passed to it and then returns the result.

y = function_name(x_1, x_2, x_3, …)

translated to C

return_type function_name (***data type*** *arg1,* ***data type*** *arg2, ….*)
{
function body
return (variable);
/* data type of variable should match with *return_type* */
}

The argument list should be a comma-separated list of data type variable name pairs. Argument values can be accessed inside the function body using these names.

Function (contd.)

- The function body consists of declaration of local variables and statements. These variables together with the function arguments can be accessed only in the function body and not outside it.
- A function is called from main or any other function as

function_name(*arg1*, *arg2*, …..);

where arg1, arg2 should be the same data type as in the function declaration.

Recursive Function

- Certain functions are defined in terms of itself. We call them **recursive functions**.
- Recall the Fibonacci number F(n) for any +ve integer n.

$$F(n) = \begin{cases} 0 & \text{when } n=0 \\ 1 & \text{when } n=1 \\ F(n-1)+F(n-2) & \text{when } n>=2 \end{cases}$$

Consider the following C code

```
int Fib ( int n )
{ if (n == 0) return (0);
  if (n == 1) return (1);
  return (Fib(n-1)+Fib(n-2));
}
```

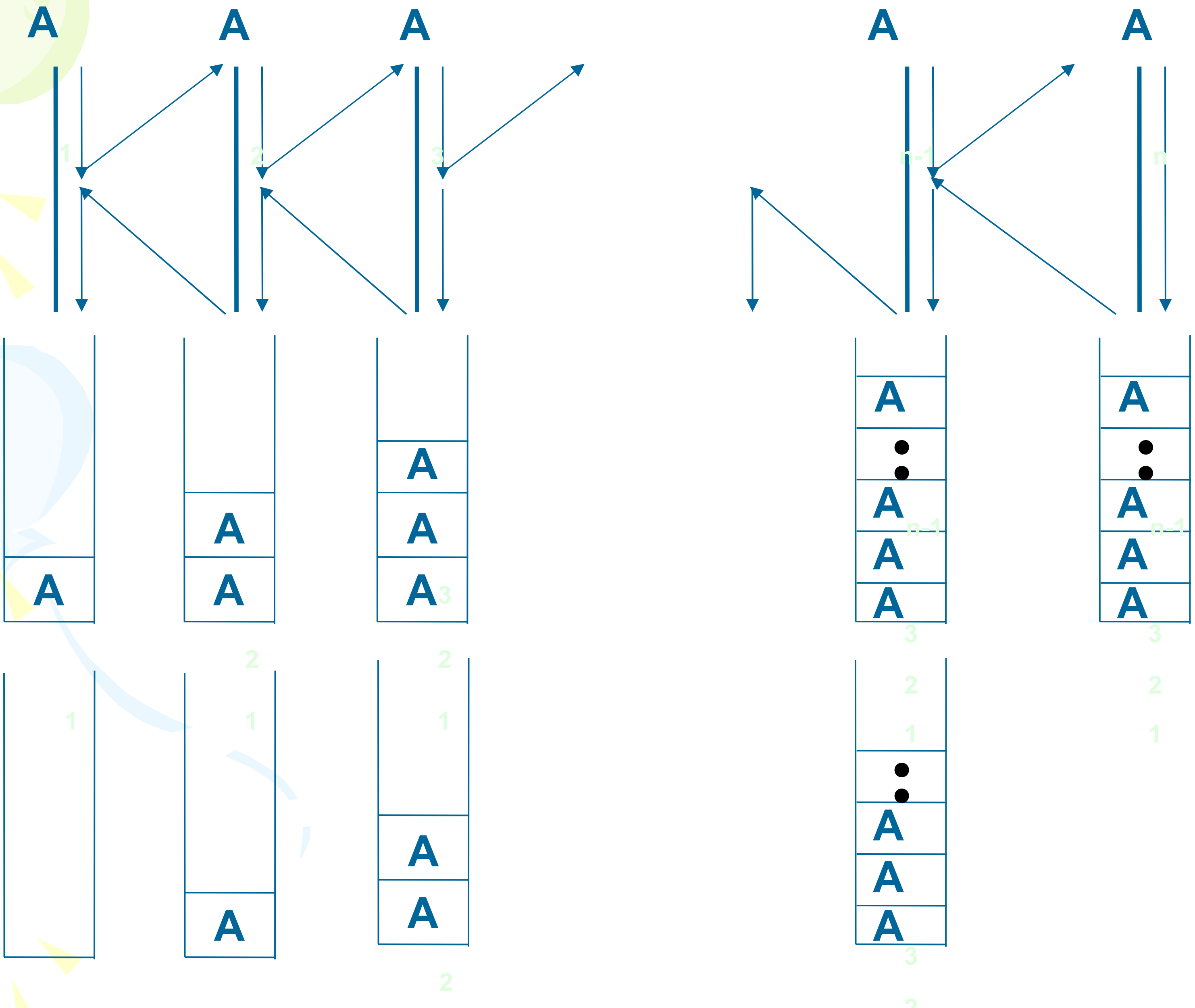
```
int main(void)
{
  int n,fib_val;
  printf("n="); scanf("%d",&n);
  fib_val=Fib(n);
}
```

Iterative version of Fibonacci

```
#include<stdio.h>
int main(void){
int n,i; /* input natural number: n and loop index: i */
unsigned int F,F1,F2; /* natural num. to store Fibonacci values*/
printf("\n Give a natural number n whose F(n) you want ::>");
scanf("%d",&n);
i = 1;    /* Initialize i to 1 */
F = 1;    /* Initialize Fi */
F1 = 0;   /* Initialize Fi-1 */
do {
++i;          /* Increment i */
F2 = F1;      /* The old Fi-1 now becomes Fi-2 */
F1 = F;       /* The old Fi now becomes Fi-1 */
F = F1 + F2;  /* Compute Fi from Fi-1 and Fi-2 */
} while (i < n);
printf("F(%d) = %d \n", i, F);
return 0; }
```

Which one is better – itearative or recursive?

Recursive Function (contd.)



Recursive Function (HW)

- Ex: Write recursive and non-recursive functions for computing the factorial of a positive integer n.

- Ex : Write a function that takes two sorted arrays and generates a combined sorted array.