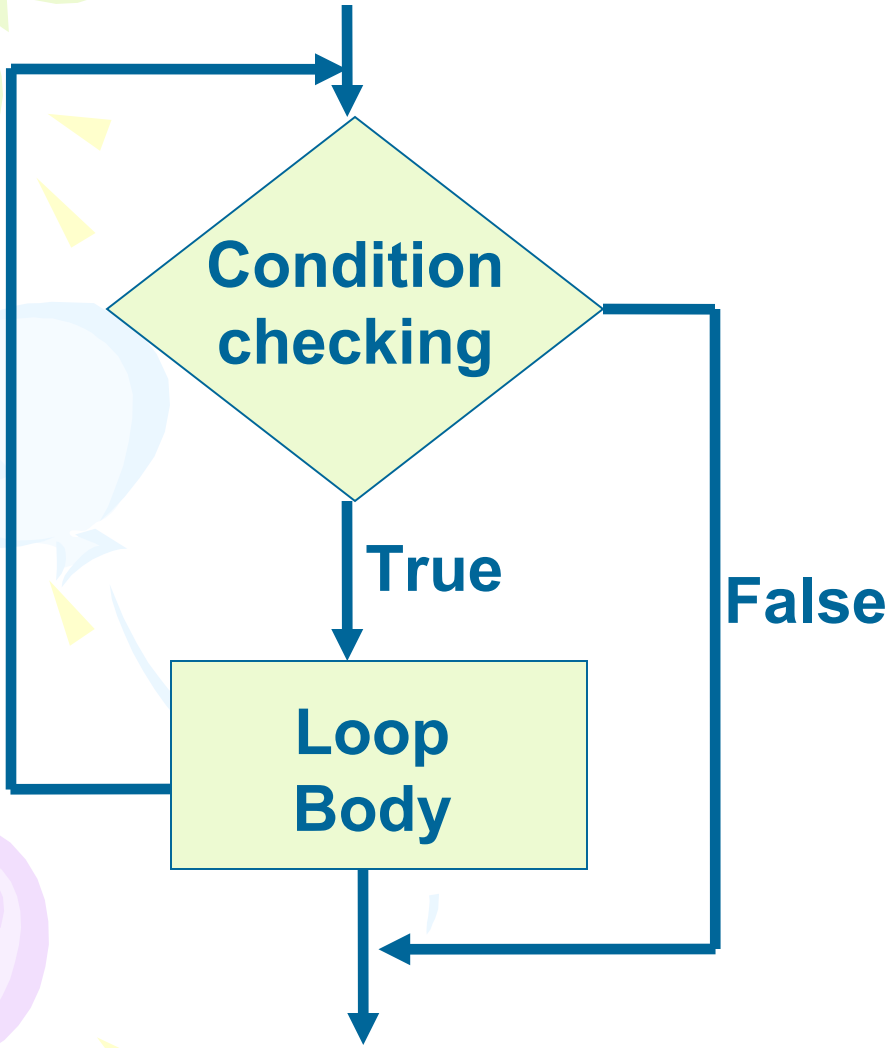
Programming & Data Structure Laboratory

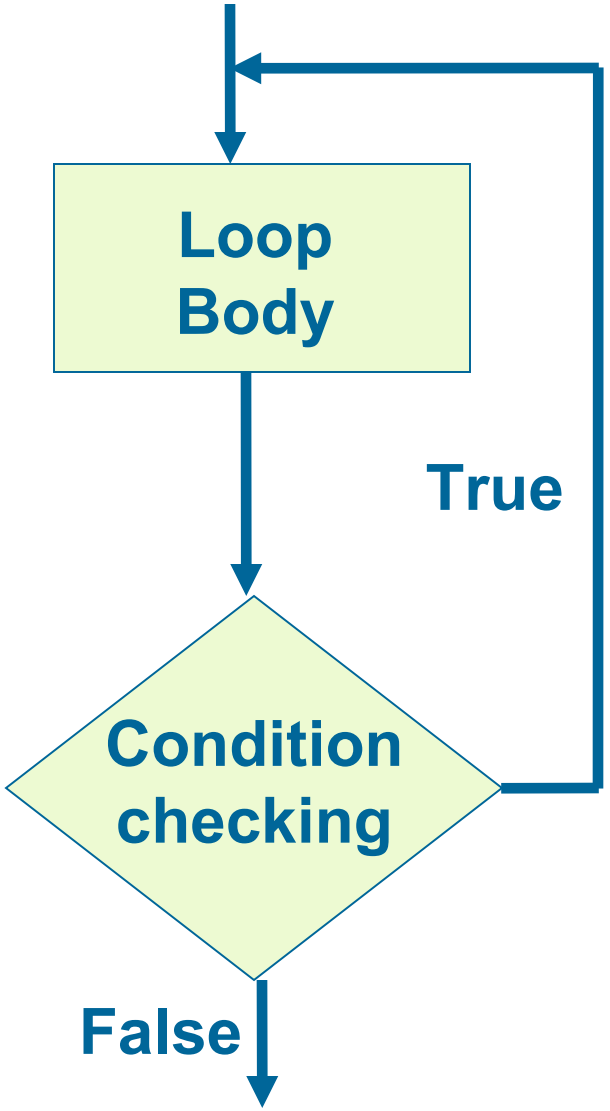
Day 2, July 24, 2014

Loops

- Pre and post test loops
- for
- while
- do-while
- switch-case

Pre-test loop and post-test loop





for-loop

```
for (initialization; condition; update)
{
/* Put in the piece(s) of code (e.g., one or more
   statements) that should be executed as long as
   the condition defined above is true */
}
```

Example:

```
int a=3;
for (i=0; i<=5; i++)
{
   a=a+a*i;
}
```

while-loop

```
expression_1;
while (condition)
{
    /* Put in the piece(s) of code (e.g., one or more
    statements) that should be executed as long as the
    condition defined above is true */
}
```

Example

```
i=0;
while (i<=5)
{
    a=a+a*i;
    i++;
}
```

do-while loop

```
do
{
    /* Put in the piece(s) of code (e.g., one or more
    statements) that should be executed as long as the
    condition defined below is true */
}
while (condition);
```

Question: What is the difference in the functionality of while and do-while?

infinite loop

- Loop condition is always true.
- C allows you to use the break statement (`break;`) to break out of the loop.
- But not good to use.
- Check for infinite loops.
- Use infinite loops only when you know what you are using it for.

An Exercise – finding gcd

- The *greatest common divisor* gcd(a,b) of two positive integers (a>=b) is the largest natural number of which both a and b are integral multiples.
- The standard gcd algorithm is based on successive Euclidean division.
- **Theorem:** [*Euclidean gcd theorem*] Let a,b be positive integers and r = a rem b. Then gcd(a,b) = gcd(b,r).
- This theorem leads to the following iterative algorithm:

  While b is not equal to 0 do

  Compute the remainder r = a rem b.

  Replace a by b and b by r.

  Report a as the desired gcd.

- **Ex:** Write a small piece of code that computes gcd using pre and post test loops

Exercises

Ex. 2: Compute the sum

n+(n+1)+….+10

for n in the range 0<=n<=10. For other values of n, an error message is to be printed till the user inputs a correct value. Do this using *switch-case*.

Ex. 3: Given an integer n, find F(n), where F(n) denotes the Fibonacci sequence. Do not use recurrence.

F(0)=0; F(1)=1; F(n)=F(n-1)+F(n-2)

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.

An Example

```
void change(int x){  x=x+8; }

int func(int x) {  x=x+7;  return x;}

int main(void){
int x=4;
printf("x=%d ", x);  What is printed?
change(x);
printf("x=%d '', x);   What is printed?
x= func(x);
printf("x=%d", x);   What is printed?
return 0;
}
```

Exercise

Write a program that goes on in a loop taking two positive integers at a time and find their gcd. The gcd should be implemented as a function

```
? gcd(?, ?){
……..
return ?;
}
int main(void){
 ……
do{ …call the function... }while(….);
return 0;}
```

Exercise

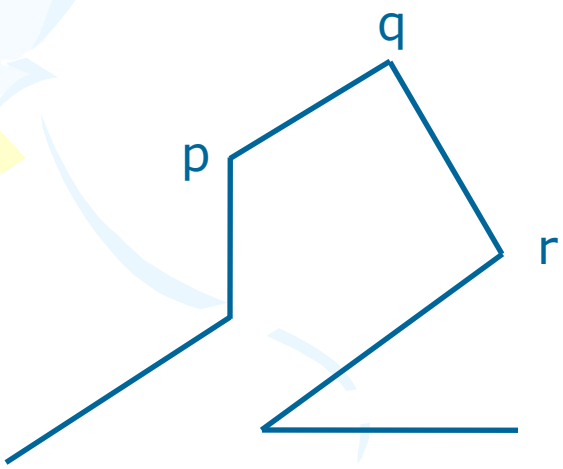
$$f(x) = \begin{cases} 2^{\wedge x} + e^{x} & \text{if } x \le 1 \\ e^{x} - 2^{x} & \text{if } 1 < x \le 2 \\ e^{x} / 2^{x} & \text{if } x > 2 \end{cases}$$

Write a C program that takes as input a real value x and computes f(x).

```
? func1(?){ … } ? func2(?){ … } ? func3(?){ … }
void caller(){ /* Takes an input x and returns the result*/}
 int main(void) { …..
 do{  caller( ); }while(flag>0);
 return 0;
}
```

Exercise

Write a program using function that first takes in two points and then goes on in a loop taking a point at a time and determines whether there was a left/right turn or a straight walk in relation to the last two points.



$$\begin{vmatrix} 1 & px & py \\ 1 & qx & qy \\ 1 & rx & ry \end{vmatrix}$$

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));
}
```

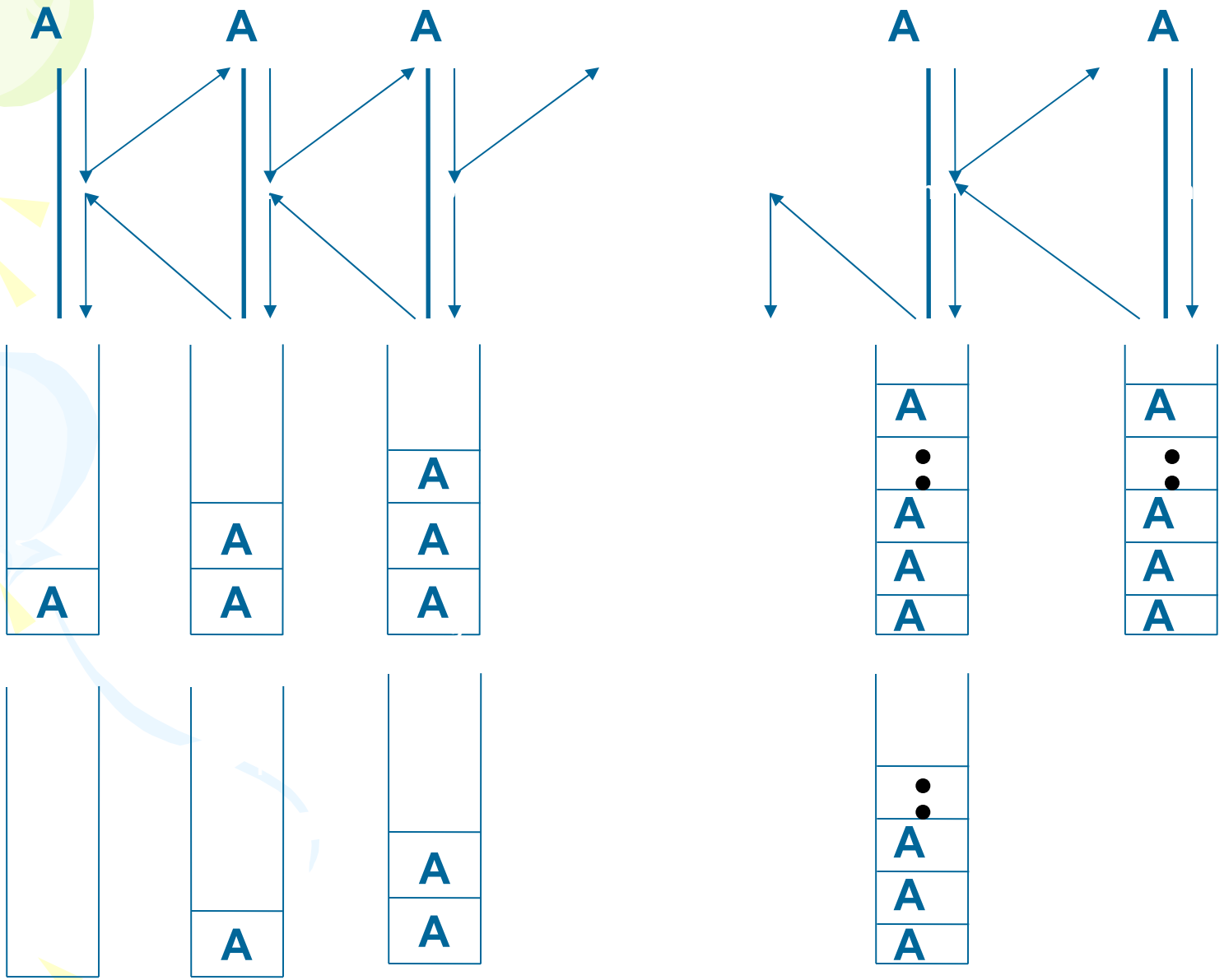
```
int main(void)
{
  int n,fib_val;
  printf("n="); scanf("%d",&n);
  fib_val=Fib(n);
}
```

Recursive Function (contd.)

```
#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?

Recursive Function (contd.)



Integer Exponentiation

- The problem is to raise a real number x to the n^{th} power, where n is a non-negative integer.
- First, write a non-recursive program to compute x^n.
- This method requires n-1 multiplications.
- Now, we will try to do better.
- Let $m = \lfloor n/2 \rfloor$, and suppose we know how to compute x^m. Then, we have two cases:

if n is even, then $x^n = (x^m)^2$,

otherwise, $x^n = x(x^m)^2$

- Now, write a recursive program to compute x^n.

```
#include<stdio.h>
#include<math.h>
double Power(double x, int m)
{  double result;
if(m==0){result=1; return result;}
else{  result=Power(x,(int)floor(m/2));   result=result*result;
if((m%2)!=0) /* If m is odd*/ result=result*x;  }
return result;}

int main(void)
{  double base,result;   int exp;
   printf("The base = "); scanf("%lf",&base);
   printf("The exponent = "); scanf("%d",&exp);
   result=Power(base,exp);
   printf("\n The result=%lf\n",result);
   return 0;}
```

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.

- 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

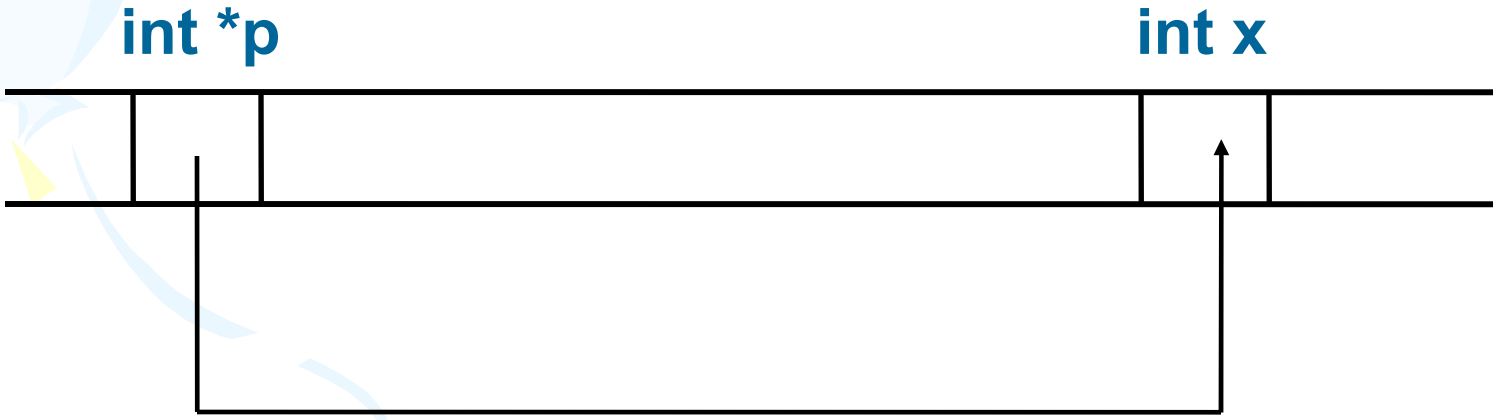
int *p

int x

p = &x

int *p

int x



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

An example - swap

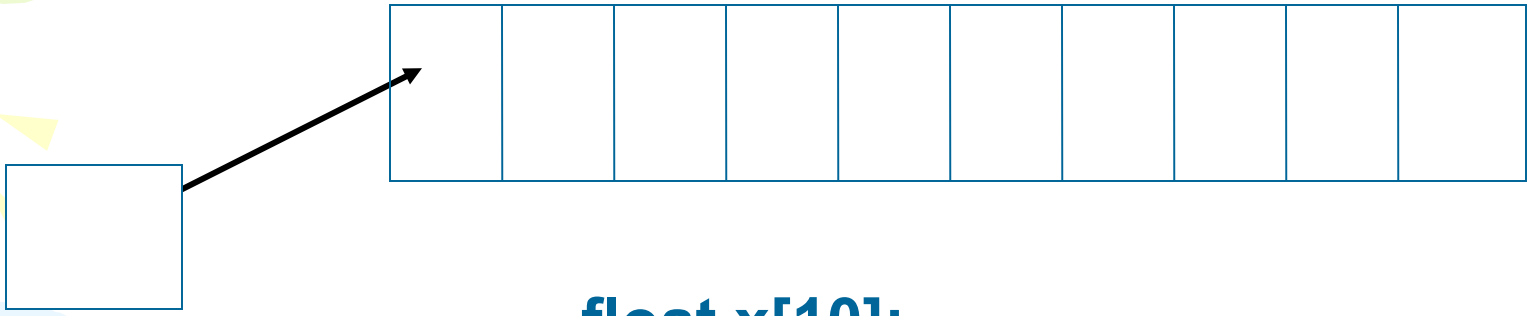
```
#include<stdio.h>
void swap(int* x, int* y)
{  int temp;
   temp=*x;*x=*y;*y=temp;
   return;
}
int main(void)
{
int a=5,b=4;
swap(&a,&b);
printf("\na=%d, b=%d \n",a,b);
return 0;
}
```

- 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];

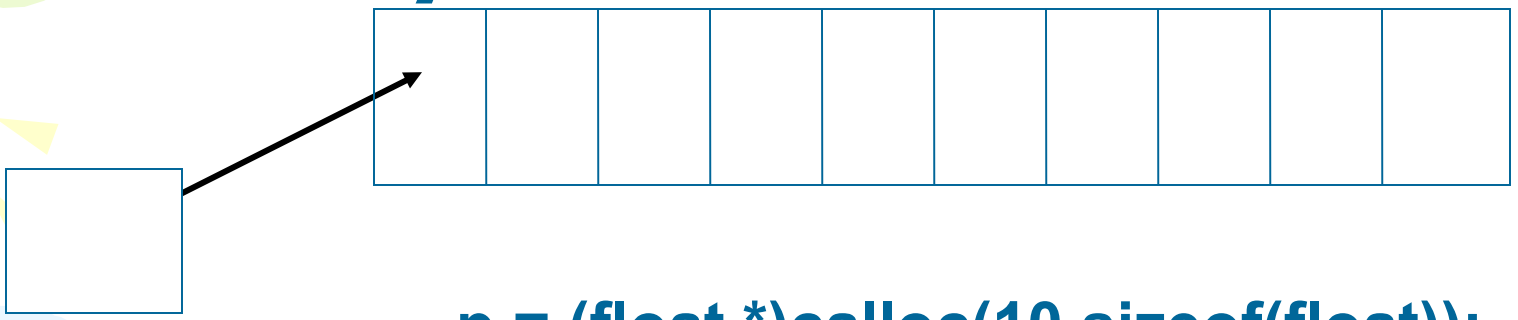
p = &x[0];

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:

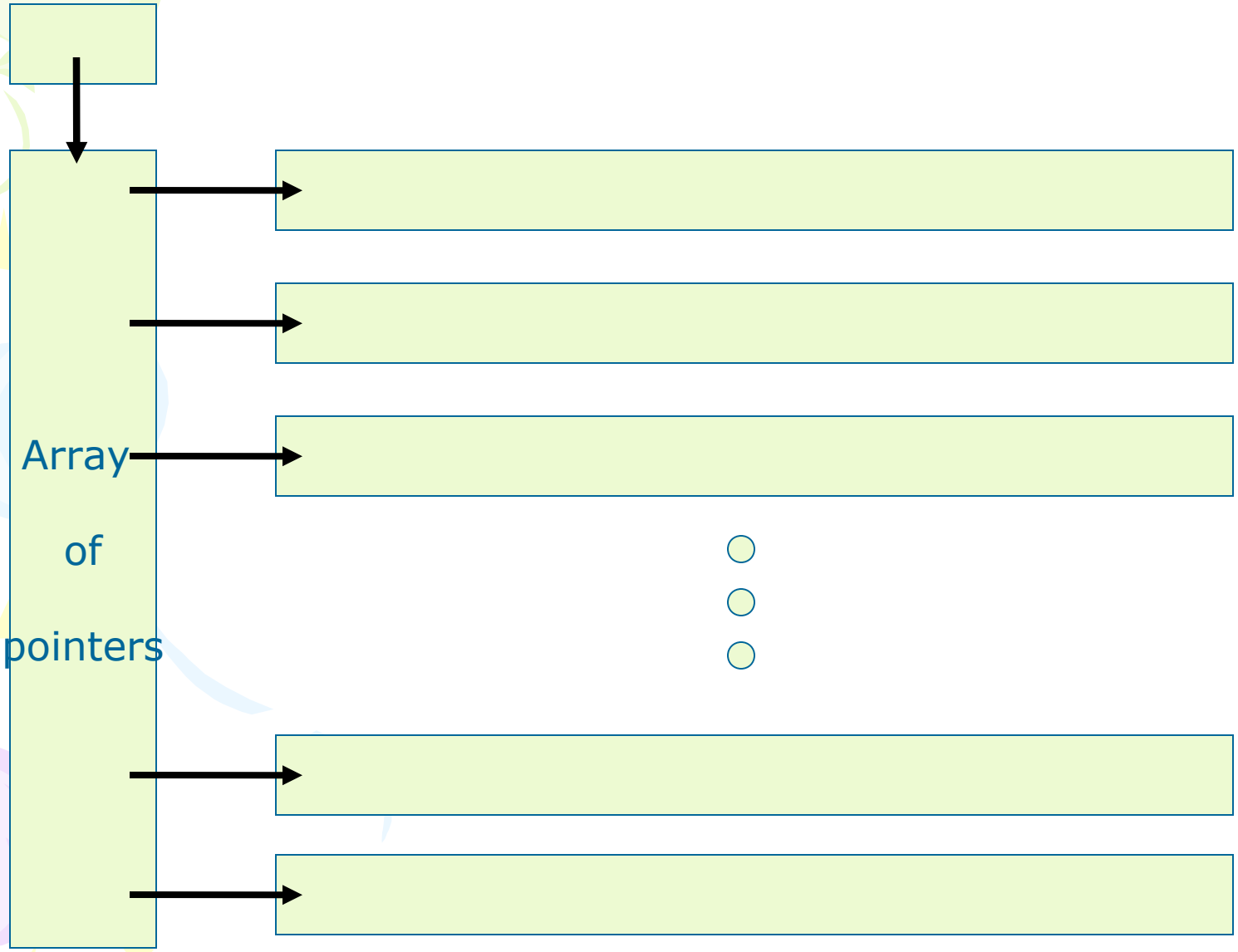
p = &x[6];

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.