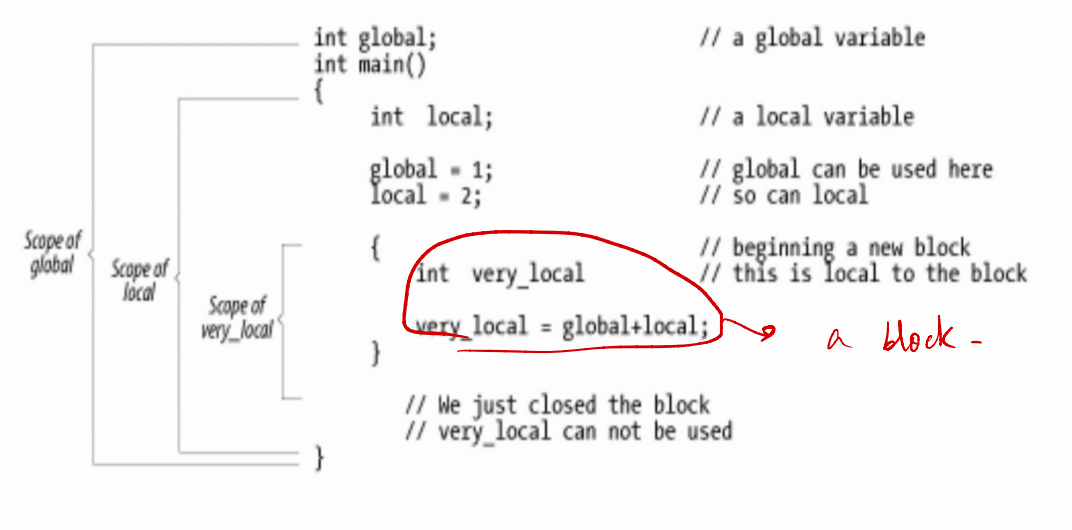
CS1111 quiz 3

by Aditya Viraj

Block, Scope, global variables



In layman terms => scope of variable is part of the code where it can be acessible.

global variable defined outside of any function generally in the start.

Block => a segment of code written between a curly brackets.

Importance of why should we use functions

To make our code less tedious.

To make the readability better.

Special functions → recursive (instead of using loops)

↓

invoking a function in another to make it less tedious.

Arguements are passed by value in C.

i.e

Values given to functions as inputs are copied to local variables.

---

functions → prototype declaration

We have specify the input type the function will take as input. no need to write a variable name

void function (int, int)

void function (int a, int b)

take longer time.

return type | function name | (type of input1, type of input2 ...)

- Void — doesn't return anything
- Int
- Char
- Can also be pointers ----- Can return Null

It is better we declare prototypes of all functions first as we may be invoking one function in an other.

Defining the function

```
int function ( int a, int b)
{
    ____ ;
    ____ ;
    ____ ;
}
```

a, b — necessary to write variable names.

Have to write code in terms of a, b as we assume they are inputs.

function invoking itself. ⇒ RECURSIVE INVOKING

Eg :-

```
int fact(int n)
{
   if (n == 1) return(1);
   return (n*fact(n-1));
}
```

base case, function doesn't do further recursive calls.

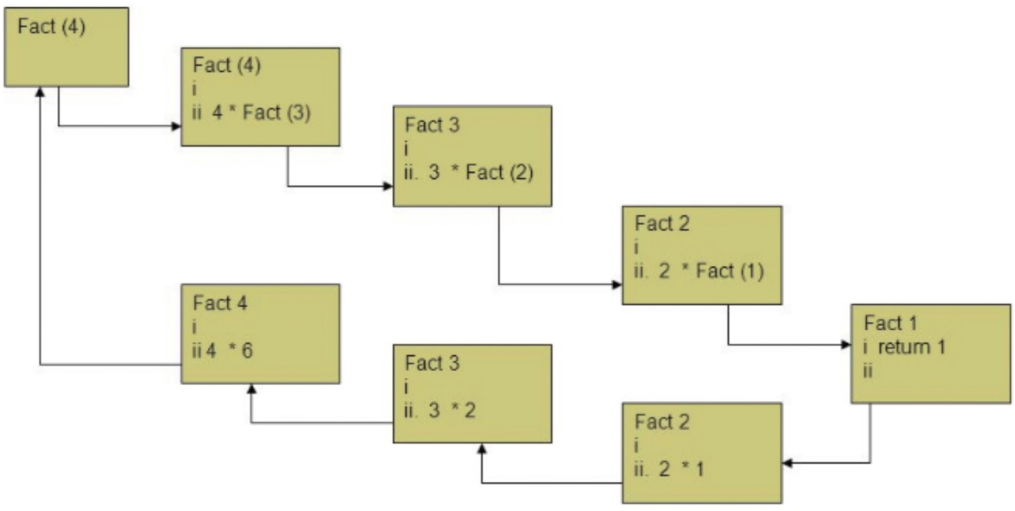
What will this function return ?!

Can have more than one base cases,

Same function with 2 bases can be modified as

```
if (n==1) return 1;
if (n==2) return 2;
:
```

A diagram to quickly recollect control flow for above recursive function at n = 4.

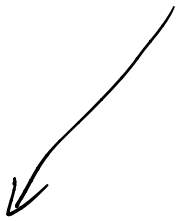


```c
#include<stdio.h>
void swap(int x, int y)
{
    int t;
    t = x;
    x = y;
    y = t;
}
int main()
{
    int a = 10, b = 20 ;
    swap(a,b);
    printf("%d %d",a,b);
}
```

⇒ What will the output be?

10 20

⇒ So to correct this we have use the concept of



Pointers → Used for storing addresses of declared variables.

declaration

type * pointer name

this is should be the data type to which the pointer points.

Eg ⇒ int * pointer , char * address int ** ptr OF ptr

can store address of an integer variable

can store address of a ptr which stores address of an integer.

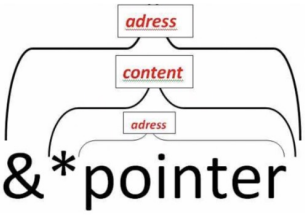
&

gives address of a variable.

*

de-referencing operator, gives variable of an address

Pointers can only be compared (>, <, ==) other operators cannot be used.



is same as pointer.

Fixing the error :

```
#include<stdio.h>
void swap(int *p1, int *p2)
{
      int t;
      t = *p1;
      *p1 = *p2;
      *p2 = t;
}
int main()
{
     int a = 10, b = 20;
     swap(&a, &b);
     printf("%d %d",a,b);
}
```

⇒ we pass address and values at those adress are swapped.

```
#include<stdio.h>
void swap(int *p1, int *p2)
{
      int *temp;
      temp = p1;
      p1 = p2;
      p2 = temp;
}

int main()
{
     int a = 10, b = 20;
     swap(&a, &b);
     printf("%d %d\n",a,b);
}
```

⇒ Doesn't acheive desired swap.

as we swap address of 'a' as $(\&b)_{initial}$ and $(\&b)_{final} = (\&a)_{initial}$

but values of a and b remain same.

int * ptr1 which is not assigned any value when dereferenced gives a rubbish random value.

If we declare an array Eg array[100].

here array is the pointer to first element.

array[i] = *(array + i)

array = &array[0]

(So if array1 = array2;
address of array2 is copied into array1.)

Incrementing/Decementing:
- advances (or moves back) the pointer to the next memory location. the exact # of byte-locations advanced (or moved back resp.) depends on the type of the pointer.

```
int a,*ptr;
 ptr = &a;
  ptr++;
```

⇒ in this cas
ptr++ has adress of a memory 4 bytes after as int takes 4 bytes.

Dynamic memory allocation ⇒ have to use <stdlib.h> header.

⇓ Used to allocate memory to variables at run time

⇓ these newly created variables become global variables.

Syntax : `ptr = (type*) malloc (byte-size)`

Example :
```
char *myCharPtr;
myCharPtr = (char *) malloc(1);
```

```
#include<stdio.h>
#include<stdlib.h>

int* ArrayCreator(int n)
{
  int *ptr;
  ptr = (int*) malloc(n*sizeof(int));
  // check if ptr is NULL to be safe
  for(int i=0;i<n;++i)
    scanf("%d",ptr+i);

  return(ptr);
}
```

⇒ To create continuous memory locations / arrays using 'malloc'.

freeing variables created by malloc is important by free(ptr) else it gets occupied and wasted.

Common error

```
int * min(int a, int b)
{
  int small;
  if (a < b)
        small = a;
  else  small = b;
  return &small;
}
```

Compile time error : returning the address of a local variable is not allowed.

Structures

This is used for creating our own new type of variable using a combination of existing and defined structures.

Also useful to store different type of variables in one continuous memory location.

Syntax

```
struct [structure tag]
   {
      member definition;
      member definition;
      ...
      member definition;
   };
```

```
struct student {
        char rollNumber[6];
        char name[20];
        int age;
        int program;
};
```

- `struct student` is a new data-type.

⇒ we have to use struct student to define a variable of such type

Eg ⇒ struct student viraj;

variable name.

We can access elements in structure ⇒ by '.'

struct . a

Eg virj . rollnumber

But we cannot check inequalities for structure

$S_1 == S_2$

Can also use a structure in a structure ⇒

```
#include<stdio.h>
struct point {
     int x_coord;
     int y_coord;
};

struct rectangle {
     struct point lower_left;
     struct point upper_right;
};
```

---

pointer to a struct ⇒ similar to a normal variable

declaring = struct student * ptr

using ⇒ *(ptr) . name / ptr -> name

-->

both '.' and '->' associate left to right.

```
r.pt1.x          (r.pt1).x
rp->pt1.x        (rp->pt1).x
(*rp).pt1.x
```

⇒ All are equivalent.

---

typedef :- assigning a new and shorter to use name to a data type

Eg :- `typedef float fraction;`

or

```
struct student {
    char rollNumber[6];
    char name[20];
    int age;
    int program;
};
typedef struct student STUDENT;
```

=

typedef

```
struct student {
    char rollNumber[6];
    char name[20];
    int age;
    int program;
} STUDENT;
```

Nodes / Self referencing Structures

used to create linked memory locations similar to arrays but not continous.

```
struct node
{
  int Data;
  struct node *Address;
}
```

→ this is address of next node.

So every non-closed link list has a head and a tail.

= Eg with 4 nodes

```
for (i=0; i<4; i++) {
  scanf("%d",value);

  newnode = (Node *) malloc(sizeof(Node));
  newnode->val = value;
  newnode->next = NULL;

  if (head == NULL) head = newnode;
  if (tail != NULL) tail->next = newnode;

  tail = newnode;
}
```

⇒ To create such a list.

Similarly we can add, delete, traverse through and search the list.

Due to stdio.h as header keyboard is the standard input and moniter is the standard output.

But we can change this and let the program take input from a text file and vice versa.

For this we have to open the file →

FILE * fopen; ⇒ stream pointer / pointer to a file.

- Prototype : `(FILE *) fopen("filename","mode");`
- Usage Eg : `(FILE *) fopen("mydatafile.txt","r");`

then use fprintf / fscanf (fopen, "...", ...);

We can also give stdin / stdout in the place of fopen.

Useful table

Precedence order	Operator	Associativity
1	() [] →	Left to right
2	++ -- - (unary) ! ~ * & sizeof	Right to left
3	* / %	Left to right
4	+ -	Left to right
5	<< >>	Left to right
6	< <= > >=	Left to right
7	== !=	Left to right
8	& (bitwise AND)	Left to right
9	^ (bitwise XOR)	Left to right
10	| (bitwise OR)	Left to right