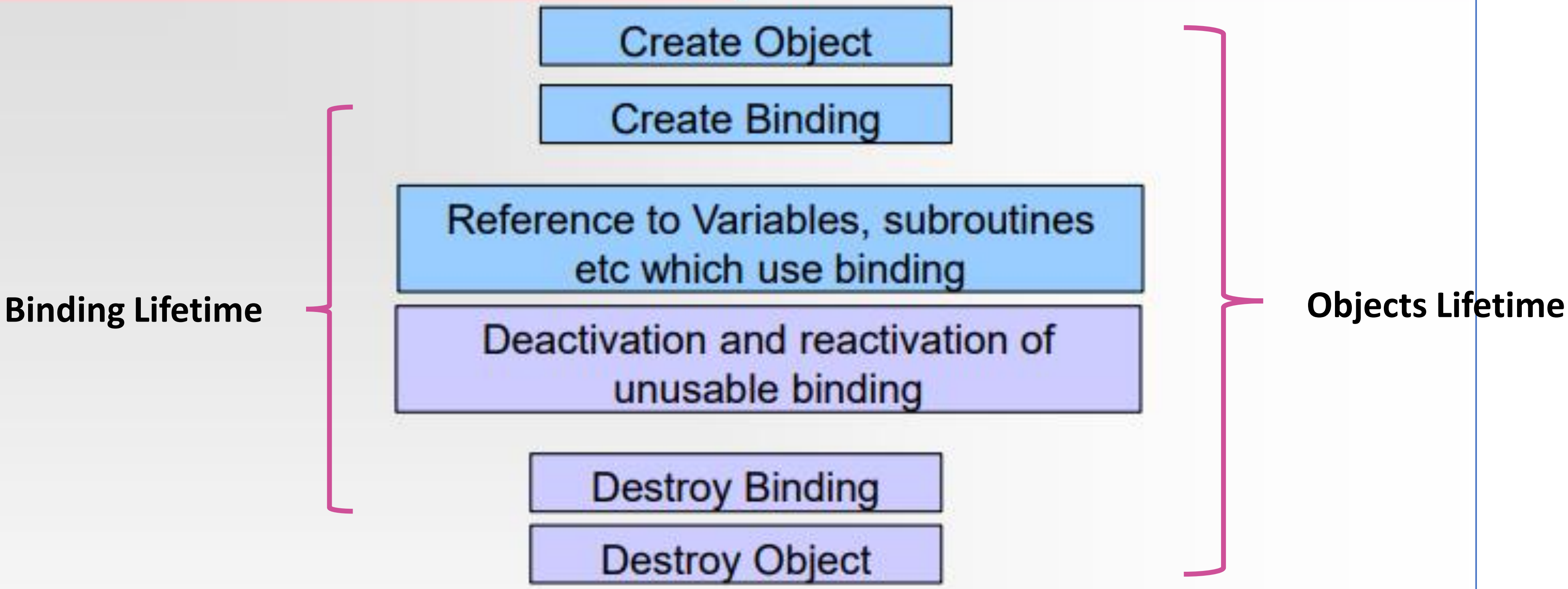
Course: PARADIGMS AND COMPUTER PROGRAMMING FUNDAMENTALS (PCPF)

Course Instructor

Mrinmoyee Mukherjee B.E (Electronics), M.E (EXTC), PhD (Pursuing)
Assistant Professor
Department of Information Technology
St. Francis Institute of Technology
email: mrinmoyeemukherjee@sfit.ac.in

Academic Year: 2023-24 (Odd Semester)

Object Lifetime



- **Binding's lifetime:**
  - The period of time between the creation and the destruction of a name-to-object binding
- **Object's lifetime:**
  - The time between the creation and destruction of an object

Stoage Allocation Mechanisms

- Object or variable lifetimes generally correspond to one of **three principal storage allocation mechanisms**, used to manage the object's space:

1. Static objects are given an absolute address that is retained throughout the program's execution.
2. Stack objects are allocated and deallocated in last-in, first-out order (LIFO), usually in conjunction with subroutine calls and returns.
3. Heap objects may be allocated and deallocated at arbitrary times. They require a more general (and expensive) storage management algorithm

Static Variables

Ex1: Java static variable

```
class Student{
   int rollno;
   String name;
   String college="ITS";
}

class Student{
   int rollno;
   String name;
   static String college ="ITS";
            //static variable
}
```

Ex2: Initialization

```
#include <stdio.h>
int main()
{
    static int x;
    int y;
    printf("%d %d", x, y);
}
```

Output: 0 and garbage

- Static variables are bound to memory cells (**not stack**) before execution begins and remains bound to the same memory cell throughout execution of program.
  - All FORTRAN variables, C static variables in functions
- Static variables are initialized as 0 if not initialized explicitly similar to global variables.
- Advantages:
  - Efficiency: All **addressing** of static variables can be **direct**. **No run-time overhead** is incurred for allocation and deallocation of static variables.
  - History-sensitive: variables retain their values between separate executions of the subprogram.
- Disadvantage:
  - Storage cannot be shared among variables.
  - Ex: if two large arrays are used by two subprograms, which are never active at the same time, they cannot share the same storage for their arrays.

Example of Static Variable

```c
#include<stdio.h>
int fun()
{
 static int count = 0;
 count++;
 return count;
}
int main()
{
 printf("%d ", fun());
 printf("%d ", fun());
 return 0;
}
```

Output: 1 2

```c
#include<stdio.h>
int fun()
{
 int count = 0;
 count++;
 return count;
}
int main()
{
 printf("%d ", fun());
 printf("%d ", fun());
 return 0;
}
```

Output: **1 1**

Stack-dynamic Variables

Recursion

```
int f(int x) {
        int a = 0;
        int y=5;
        ...
        f(y+a);
        ...
    }
```

- A stack-dynamic variable is one that is bound to an address on the stack, and created dynamically for that purpose
- It may also be unbound during run-time, and its memory cell deallocated by being popped off the stack.
- Stack-dynamic variables are allocated from the run-time stack.
- At any time during the run of the program, the stack contains the memory cells for all the subroutines currently executing, including all the invocations of recursive subroutines currently executing.
- Advantages:
  - Allows recursion: each active copy of the recursive subprogram has its own version of the local variables.
  - In the absence of recursion, it **conserves storage** b/c all subprograms share the same memory space for their locals

Example Subroutine Calls

Function-1

```
int max(int num1, int num2) {
  int result;          /* local variable */
  if (num1 > num2)
    result = num1;
  else
    result = num2;
  displayOutput(result);
  return result;
}
```

Function-2

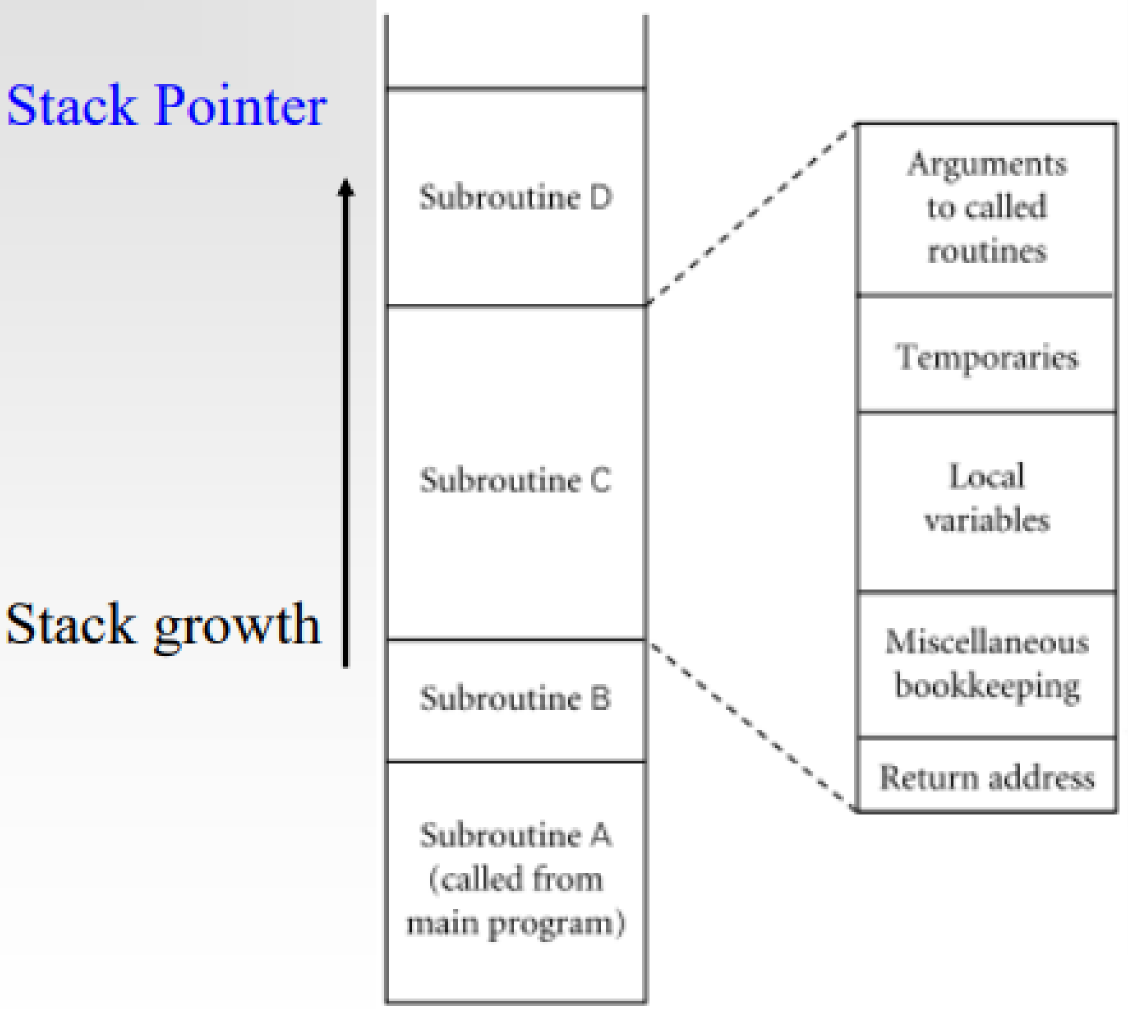
```
displayOutput(int x)\{
          -----
}
```

```
int main () {

  /* local variable definition */
  int a = 100;
  int b = 200;
  int ret;

  /* calling a function to get max value */
  ret = max(a, b);

  printf( "Max value is : %d\n", ret );

  return 0;
}
```

Mrinmoyee Mukherjee

Stack-dynamic Variables in Subroutine

Subroutine A calls subroutine B and B calls subroutine C
and C calls subroutine D



Subroutines

```
int f(int x) {
        int a = 0;
        int y=5;
        ...
        f(y+a);
        ...
    }
```

Recursion

```
int f(int x) {
        int a = 0;
        int y=5;
        ...
        f(y+a);
        ...
    }
```

The maintenance of the stack is done by subroutine calling sequence, prologue and epilogue

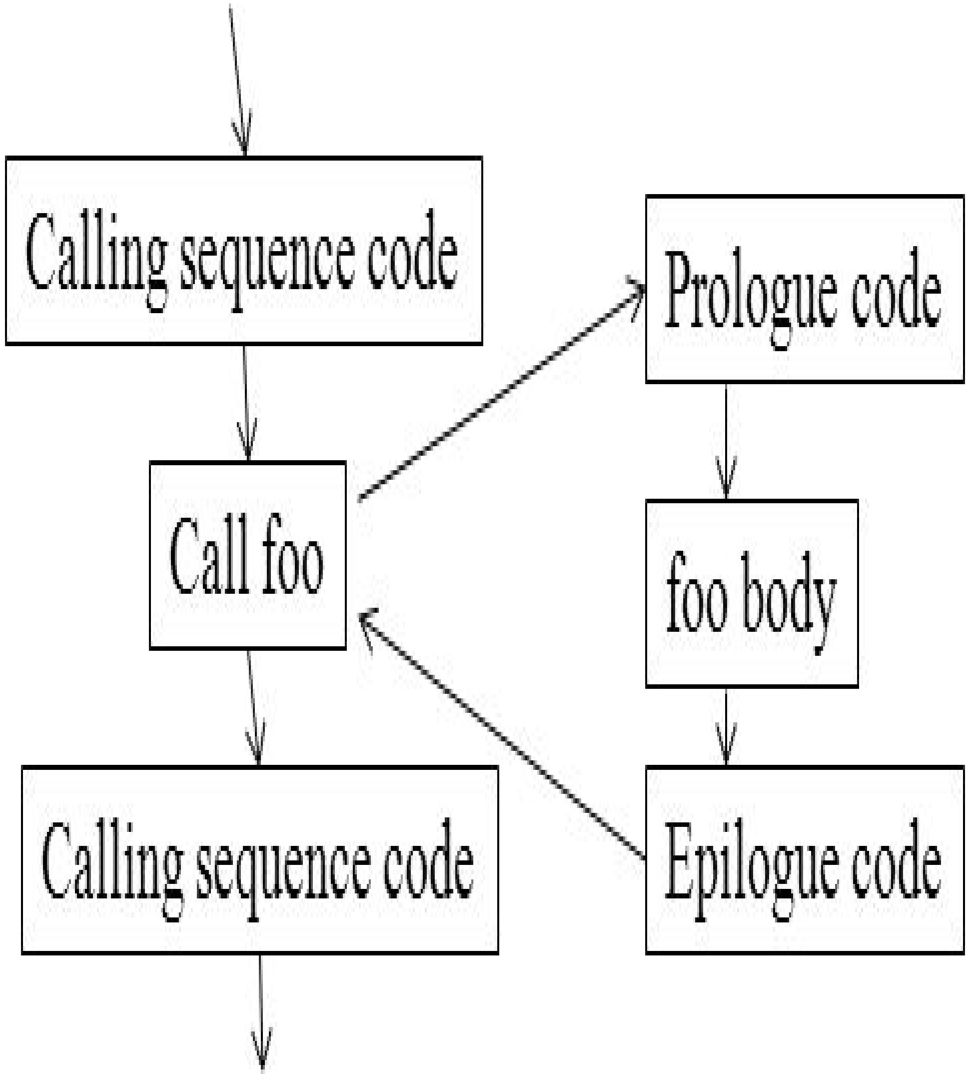
Calling Sequence: Code executed by the caller immediately before and after the call

Prologue: Code executed at the beginning

- Allocates a frame by subtracting frame size from sp
- Saves callee-saves registers used anywhere inside callee

Epilogue: Code executed at the end

- Puts return values into registers
- Restores saved registers using sp as base
- Adds sp to de-allocate frame



Mrinmoyee Mukherjee

Explicit Heap-dynamic Variables

- The heap is a unstructured pool of nameless memory cells.
- variables are bound to memory cells that are allocated and deallocated by explicit run time instructions specified by programmer during execution.
- lifetime = from explicit allocation to explicit deallocation
- These variables, which are allocated to and deallocated from the heap, can only be referenced through pointers or reference variables.
- The heap is a collection of storage cells whose organization is highly disorganized because of the unpredictability of its use.
- e.g. Dynamic objects in C++ (via new and delete)

```
int *intnode;                 // create a pointer
intnode = new int;            // allocates the heap-dynamic variable
delete intnode;     // deallocates heap-dynamic variable to which intnode points
```

  - An explicit heap-dynamic variable of int type is created by the new operator.
  - This operator can be referenced through the pointer, intnode.
  - The var is deallocated by the delete operator.

What is malloc()

malloc is a built-in function declared in the header file <stdlib.h>

malloc is the short name for "memory allocation" and is used to dynamically allocate a single large block of contiguous memory according to the size specified.

Syntax:

```
(void* )malloc(size_t size)
```

malloc function simply allocates a memory block according to the size specified in the heap and on success it returns a pointer pointing to the first byte of the allocated memory else returns NULL.

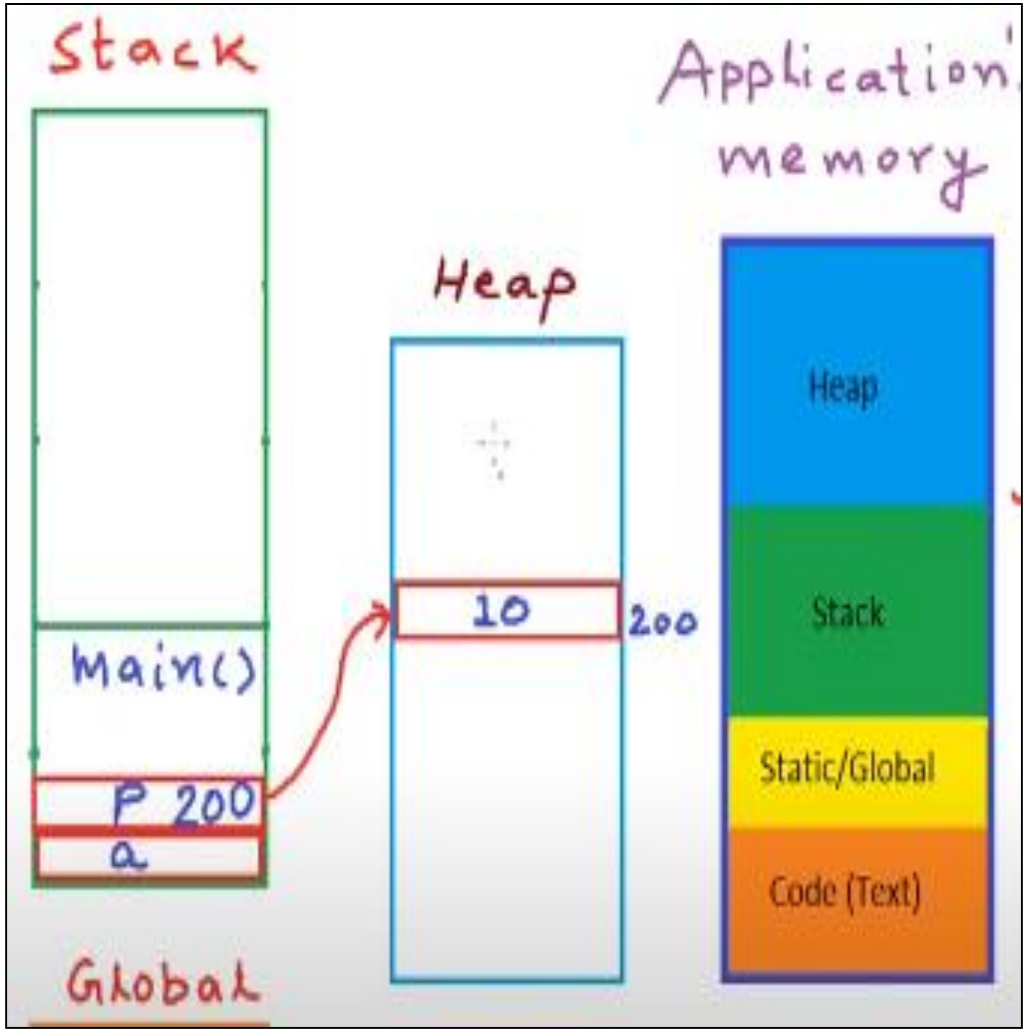
The void pointer can be typecasted to an appropriate type.

```
int *ptr = (int* )malloc(4)
```

HEAP

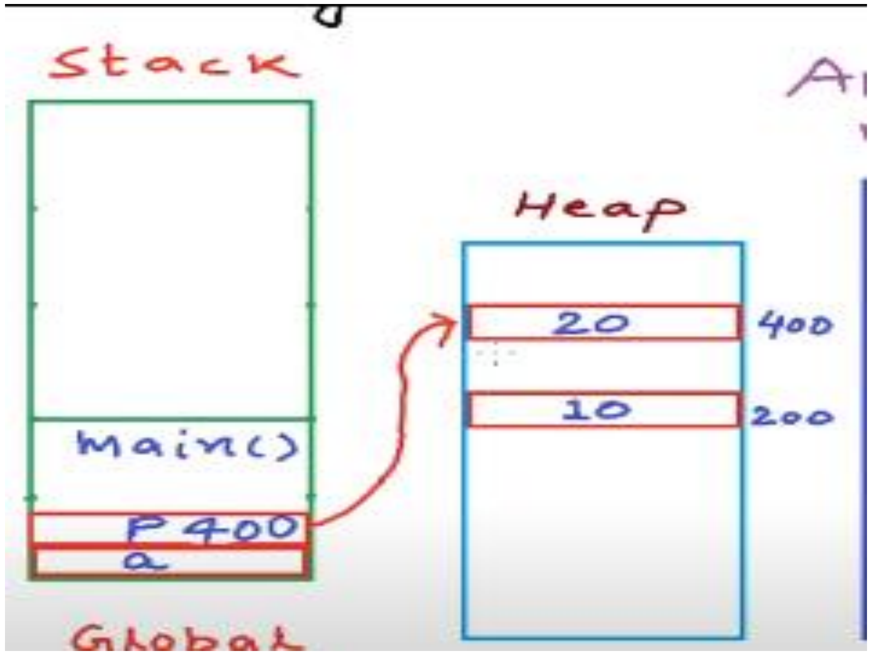
- Heap is a region of memory (storage) in which sub-blocks can be allocated and de-allocated at arbitrary time.
- Required for dynamic allocation of memory (at run time)

```
#include<stdio.h>
#include<stdio.h>
int main()
{
int a; //goes on stack
int *p;
p=(int*)malloc(sizeof(int));
*p=10;
```

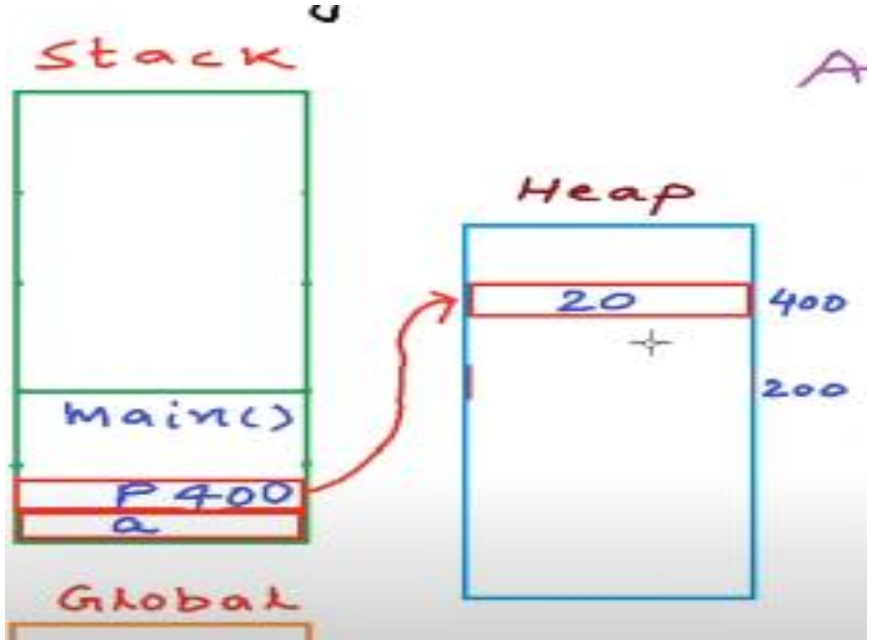


malloc()
realloc()
free()

```
#include<stdio.h>
#include<stdio.h>
int main()
{
int a; //goes on stack
int *p;
p=(int*)malloc(sizeof(int));
*p=10;
p=(int*)malloc(sizeof(int));
*p=20;
```




```
#include<stdio.h>
#include<stdio.h>
int main()
{
int a; //goes on stack
int *p;
p=(int*)malloc(sizeof(int));
*p=10;
free(p)
p=(int*)malloc(sizeof(int));
*p=20;
```



References

1. Michael L Scott, " Programming Language Pragmatics", Third edition, Elsevier publication (Chapter-3)
2. Ravi Sethi, " Programming Languages-concepts and constructs", Pearson Education (Chapter-3,4,5)

Web Resources

1. NPTEL Online Video resources- Lecture-02, Lecture-03, Lecture-10
http://www.nptelvideos.in/2012/11/principles-of-programming-languages.html

2. Stanford University Online lectures- Lecture-02 and Lecture-03
https://www.youtube.com/watch?v=Ps8jOj7diA0&list=PL9D558D49CA734A02

3. Neso Academy- Static and Dynamic Scoping (Part I and II)
https://www.youtube.com/watch?v=L53nqHCSSFY&t=52s