C Programming Lecture 12

Saturday, 22 June 2024 8:17 PM

Complex declarations

Read inside out ★★

Operator	Precedence	Associativity
(), []	1	Left to right
*, identifier	2	Right to left
Data type	3	-

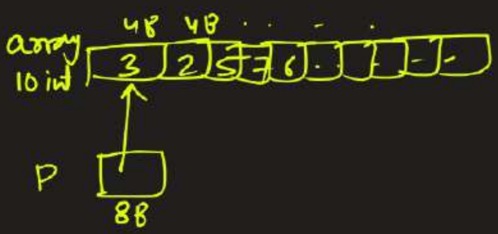
Arrays
Pointers
Functions

() [] →

* identifier ←

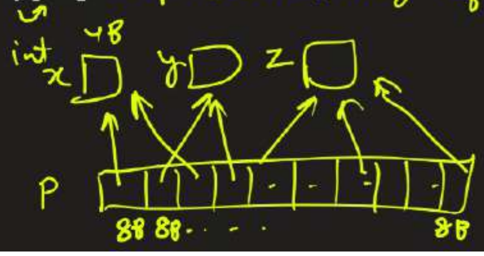
Data type

`int (*p)[10]` → p is a pointer to array of 10 integers



sizeof(p) = sizeof(pointer)
= 8B

`int *p[10]` → p is an array of 10 integer pointers



sizeof(p) = 80B

8B 8B 8B

`void *f()` → f is function which takes no arguments & returns a void pointer

`char (*f)()` → f is a pointer to a function which takes no arguments and returns a char

```
char f1 ( ){
  //logic
  return 'c';
}
```

f1 $\boxed{100}$

f $\boxed{100}$

8B

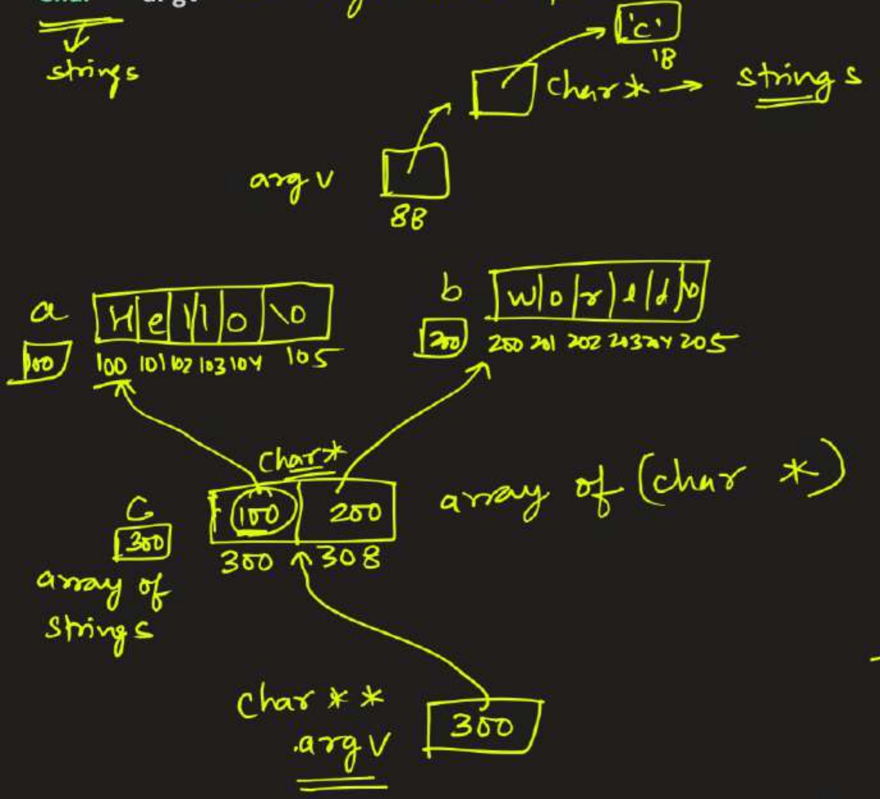
size of (f) = 8B

char **argv → argv is a pointer to character pointer

strings

argv[0] → char *

argv[1] →



Hello ✓ → a

printf ("%s", c[0]);

printf ("%s", c[1]);

World. ✓ → b

printf ("%s", a);

printf ("%c", c[0][0]);

c[0][0]

$*(*(c+0)+0)$

$= *(100+0)$

$= *100$

= 'H'

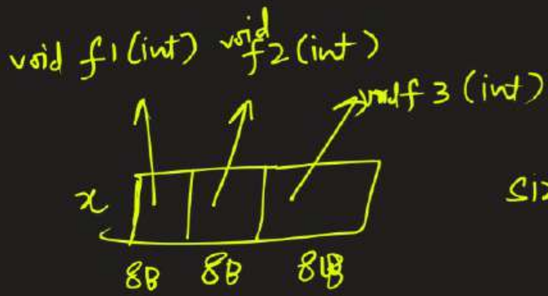
char * * argv

c[0] = *(c+0) = *c = 100 = a

printf ("%s", argv[0]); → Hello

printf ("%s", argv[1]); → World.

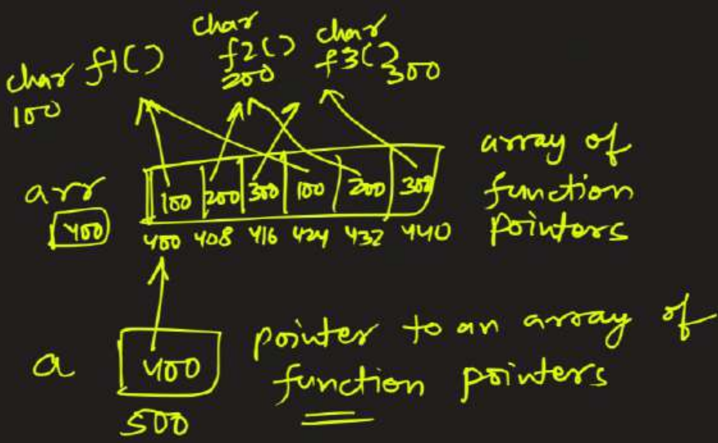
`void (*x[3])(int)` :- x is an array of 3 pointers to functions which take int as argument and return void



sizeof(x)= 24

`char (*(*f())[])()` f is a function which has no arguments and returns a pointer to an array of pointers to functions which takes no argument and returns char

```
char (*(*f())[])() {
    // logic

    return a;
}
```



}

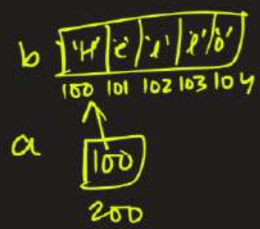
`char (*(*f[3])())[5]` → f is an array of 3 pointers to functions which have no arguments and returns a pointers to array of 5 characters

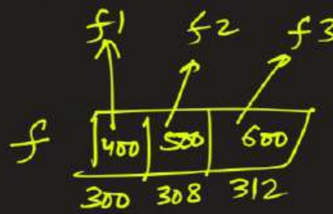
— f1 ( ) {   400

// logic

returns a;

}





sizeof(f) = 24 B

`int *(*(*f[5])())()` :- f is an array of 5 pointer to functions which have no argument and return a pointer to a function with no arguments and return an integer pointer

`int (*p)(int (*)[2], int (*)(void))` := p is a pointer to a function which takes A and B as arguments and returns an integers

`int (*)[2]` ↓ pointer to array of 2 integers (A)

`int (*)(void)` ↓ pointer to function with no arguments and returns an integer (B)

Example:

Declare the following statement:

"An array of 4 pointers to chars". ✓

```
A.  char *ptr[4]();      × invalid
B.  char *ptr[4];        valid ✓
C.  char (*ptr[4])();    ✓ valid
D.  char **ptr[4];       ✓valid
```

char (* ptr[4])( );

Char *(*ptr[4])( )

Example:

Declare the following statement?

"A pointer to an array of three chars".

```
A.  char *ptr[3]();      invalid
B.  char (*ptr)*[3];     invalid
C.  char (*ptr[3])();    ptr an array of 3 pointes to functions with no arg & return char
                         valid
D.  char (*ptr)[3];      valid ✓
```

Example:

Declare the following statement?

"A pointer to a function which receives nothing and returns a pointer to an integer".

A. `int **(ptr)*int;` invalid

B. `int **(*ptr)()` valid ×

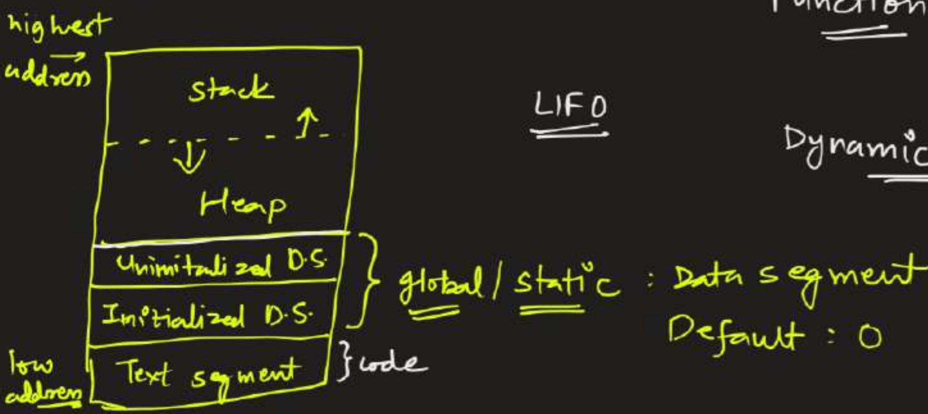
C. `int *(*ptr)(*)` invalid

D. `int *(*ptr)()` valid ✓

Memory Layout of C programs

A typical memory representation of a C program consists of the following sections.

1. Text segment (i.e. instructions) ✓
2. Initialized data segment ✓
3. Uninitialized data segment (bss) ✓
4. Heap ✓
5. Stack ✓



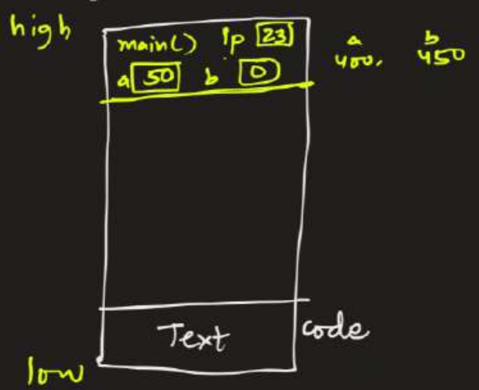
LIFO

Function → local variable, instruction ptr } activation Record → Stack

Dynamic memory → Heap

```
1  #include<stdio.h>
2  void modifyValue(intx, inty){
3      x = x * y;
4      y = y / x;
5  }
6  void modifyReference(int*x, int*y){
7      *x = *x * (*y);
8      *y = *y / (*x);
9  }
10 void swap(int*a, int*b){
11     int temp = *a;
12     *a = *b;
13     *b = temp;
14 }
15 void process(int*x, int*y){
16     modifyValue(*x, *y);
17     swap(x, y);
18     modifyReference(x, y);
19 }
20 int main(){
21     int a = 5, b = 10;
22     process(&a, &b);
23     printf("a = %d, b = %d\n", a, b);
24     return 0;
25 }
```

global / static variables ×



local → default garbage

stack

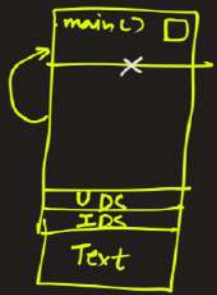
ip 17

activation records get pushed in the stack = 5

m, process, mV, swap, mref

Local Variables

```c
#include<stdio.h>

int func(){
auto int count=0;
    count++;
    return count;
}

int main(){
  printf("%d \n",func());   1
  printf("%d \n",func());   1
  return 0;
}
```



★ Function call gets replaced by the returned value

Static Variables

```c
#include<stdio.h>
int func(){
  static int count=0;   ignored
  count++;
  return count;
}

int main(){
  printf("%d \n",func());   1
  printf("%d \n",func());   2
  return 0;
}
```

static + global

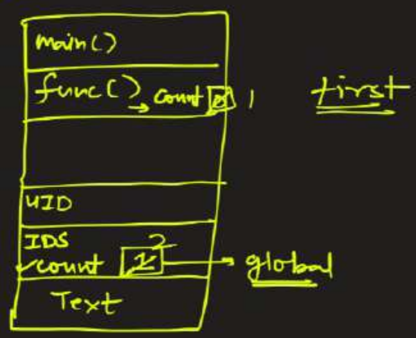
Static variables get initialized only once, they reside in the data segment, and their default value is 0.

}

Static variables get initialized only once, they reside in the data segment, and their default value is 0.

Global Variables → visible to entire program.

```
#include<stdio.h>
int count = 0;

int func(){    int count=0;
  count++;
  return count;
}

int main(){
  printf("%d \n",func());  1
  printf("%d \n",func());  2
  return 0;
}
```

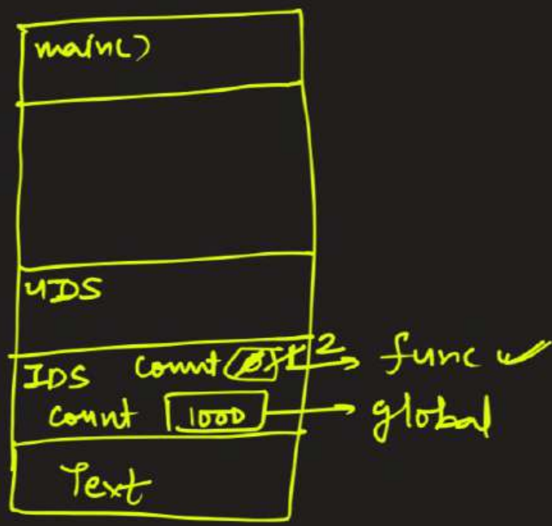


* Compiler first looks for a variable in stack/activation record (local variables), then it looks for the variable in data segment but visible only to that function (static variable), then it looks for global variables in data segment.

```c
#include<stdio.h>
int count = 1000;
int func(){
  static int count = 0;
  count++;
  return count;
}

int main(){
  printf("%d \n", func());   // 1
  printf("%d \n", func());   // 2

  printf("%d \n", count);    // 1000

  return 0;
}
```

local → static → global

local → static → global
× ×



=x=