* Say: (Chaos *) → does loop → say int: not making all zeros!

* Just no longer pointing! (freed)!

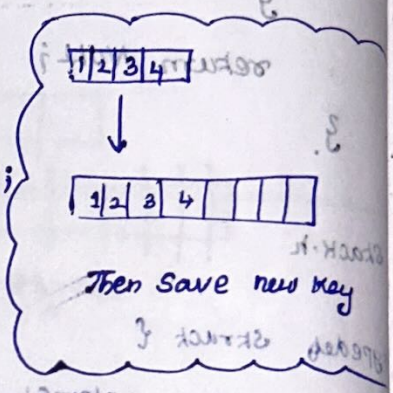
---

* dynamic c strings → Need to make sure all are disposed properly

* Int specific: don't worry! (does loop: dispose all: no need)

```
void StackPush(Stack *S, int value)
{
  if (S->logLength == S->allocLength)
  {
    S->allocLength *= 2;
    S->elems = realloc(S->elems, S->allocLength
                       * sizeof(int));
    assert(S->elems != NULL);
  }
  S->elems[S->logLength] = value;
  S->logLength++;
}
```

Check: Saturated, reallocate.



Then Save new key

---

c++ → has no realloc equivalent!

---

* dynamically allocated figure: enough memory — Just extend — same exact address

* else move Somewhere, extend, copy! → free, find new, copy! → different one! (returns) address

---

realloc(S->elems, S->allocLength * sizeof(int));

↓

Forget to assign [Stu preferred to old: may be changed — Just dead memory].

What happens (theory): realloc: fails: can't return NULL → but original address - without reallocating.

① original: NULL pointer,

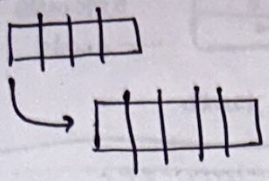
Now: Fail (returns org. address) — NULL

Point: error message: realloc failed!

---

free knows, calloc, malloc: knows how many bytes remaining!

realloc rarely called: 4 → 1 time, 8 → 1 time, 16 → 1 time
for every 2^N time. (So more as size ↑)



Trick: Interpretation same
(integer alone moving)!
base! (preserve) — Same order
Same pattern!

* Time Consuming! — actually only copies all data
* O(N) actually → Important point is search heap for free space!

int StackPop (Stack *s)
{
assert (s->logLength > 0); → go ahead do
s->logLength--;
return (s->elems[s->logLength]);
}

most recent element — removed

reallocs: ignores any request shrink an array (as long as size request is not) — doesn't even care! (won't more get more!)

generic

Stack pop: Return value: 4byte figure (int)] easy to do
8byte: double

But — I want generic — void *

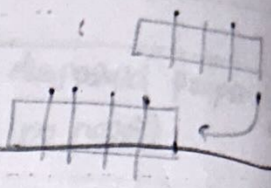
Problem: Stack *s;
StackNew(s);
→ uninitialized! "where to store?" — Core dump

Solution
Stack s;
StackNew(&s);

generically!

handle: ints, bools, doubles, Stucks, Chars * → Challenge: (dynamically allocated memory)!

```
typedef struct {
    void *elems;
    int elemSize;
    int logLength;
    int allocLength;
} stack;
```



Lost: pointer arithmetic without char* casting!

Knowledge about how big my data type is!

(can't assume as int anymore!

Challenge 1: how I'm going to initialize? - stack

* elemSize → So we know - how long!

*

```
void StackNew(stack *s, int elemSize);
void StackDispose(stack *s);
void StackPush(stack *s, void *elemAddr);
void StackPop(stack *s, void *elemAddr);
```

→ Initialize elemSize.

* why void * elemAddr → I don't know: It may be int/...

* StackPop → Supply address: previous element → easy to remove!

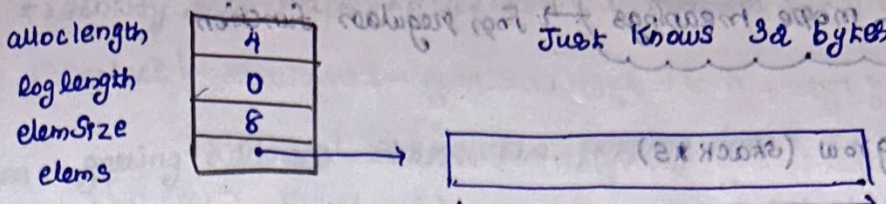
* Rather than using assignment → rely on mem Copy

Stack Copy:

```
void StackNew(stack *s, int elemSize)
{
    assert(s->elemSize > 0);
    s->logLength = 0;
    s->allocLength = 4;
    s->elemSize = elemSize;
    s->elems = malloc(4 * elemSize);
    assert(s->elems != NULL);
}
```

→ True (stopped!) → Not important!

I don't know datatype but elemSize



```
void StackDispose (stack *s)
{
    free (s->elems);
}
```

→ Incomplete (without knowing: Chaos** desaster)!

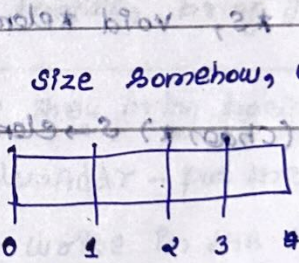
```
void StackPush (stack *s, void *elemAddr)
{
    if (s->logLength == s->allocLength)
        StackGrow (s);

    void *target = (char*)s->elems + (s->logLength * s->elemSize);
    memcpy (target, elemAddr, s->elemSize);
    s->logLength++;
}
```

→ Takes care of growing job!

Job: take void *elemAddr size somehow, write in next slot!

---

3 elements filled




```
(char*)array + (s->logLength) * (s->elemSize);
```

→ address need to given for next one!

```
memcpy(array pointer, element pointer, elemSize);
       dest           source          (int)
```

"can't do pointer arithmetic on void *"

Tricks:

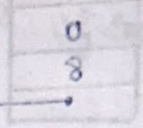
Tech 1 * cast (char*) → pointer arithmetic → add bytes!

Tech 2 * (unsigned long) → do normal arithmetic!

Plain math

long - size of the word on a register set.

("static" — more meanings) → For regular function




```
static void stackGrow (stack *s)
{
    s->allocLength *= 2;
    s->elems = realloc (s->elems, s->allocLength * s->elemSize);
}
```

Meaning stackGrow → private function → shouldn't be accessed outside this file (private in C++ sense!)

* static → makes this file: internal linkage

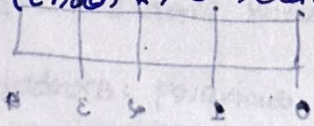
used internally by .o files

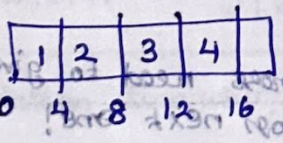
stackPush, ... are all global functions - accessible from other .o files!

why: each one writing a swap (global) → collision!

Linker: which one? (make it internal!)

```
void StackPop (stack *s, void *elemAddr)
{
    void *source = (char *) s->elems + (s->logLength - 1) *
                                        s->elemSize;
```





Now 4 elements

Remove 4th one → go in to its address

```
    s->logLength--;
```

(or) directly like above!

```
void StackPop (stack *s, void *elemAddr)
{
    void *source = (char *) s->elems + (s->logLength - 1) *
                                        s->elemSize;
    memcpy (elemAddr, source, s->elemSize);
    s->logLength--;
}
```

* Identify ~~Safe~~ place to write 1 element
* Replace!

Idea: getting address - store the popping data.
(give your suitcase - basket, here what you asked).

Why! dynamically shrink: Calloc, malloc, realloc.

> Note: Void * return → pointed to a dynamically allocated array element.

Convention: * Don't like a function dynamically allocate & make it the responsibility of user of that func. to free it!
(Asymmetry of responsibility)

make: when func allocates, itself - habit! (Symmetry - maintain).

Say: Void * → each time I'm poping I'm giving a pointer to dynamic array! - lot of pointers - Future: maynot exist: Changed - ~~hard~~ hard to maintain?

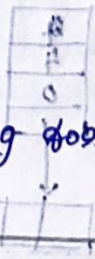
Say: we don't give you basket, bring your own - get it!

---

* Now: As they bring basket (& pop)
* void * elemAddr - put the item in their basket
* meaning: write in the memory - elemAddr pointing!

How to Know: popping worked Correctly?

* unit tests: written on .c
written as a client



Say 1 to million - write pop! (check using for loop!)
1 million to 0 (i = pop value)
Fails: Something wrong!

Say: Testing: instead of (twice realloc)
realloc every single time

why? test: everything working fine!

dealing with generics, memcpy, void * → much more an expert!

* C++ → strongly type - more type checking - templates — But not C.
* easy to do mistakes: why: forget elemSize in realloc (Low bytes - crash)!

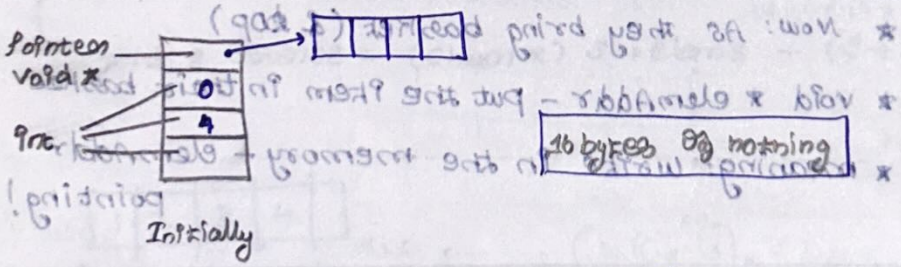
Lecture: 7

Strings! → Collection → print → In reverse order!

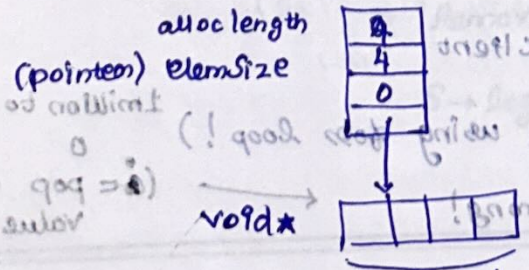
```
int main (__, __)
{ const char * letters[]={"Al", "Bob", "carol"};
  Stack stringStack;
  StackNew (&stringStack, sizeof (char*));
  for (int i=0; i<3; i++) {
      char * copy = strdup (friends[i]);
      StackPush (&stringStack, &copy);
  }
```

dynamically allocated c strings!

'next page: use pop' - get the name to print!



After StackNew

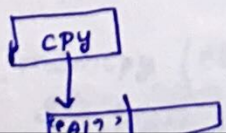


Job: copy - each string in the resp. pointers

1st iteration

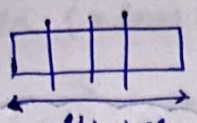
In stack - push

ownership is transferred to stack



'call by reference'

What's the second argument: stackpush



(doesn't know it is char* → 4 bytes each)

So what we can do → cpy address of string in each 4byte boxes!

→ what I need to do: pass address of 1 element of string! (char*)

→ meaning: copy [which has address].

Wait a minute:

memcpy(destpointer, sourcepointer, size);

Data is copied → not pointer!

memcpy(&, copy, size);

↓

pointer

Copy has data: say: A in 1st iteration

I want address to be copied!

Say & copy → passed [Address of pointer]

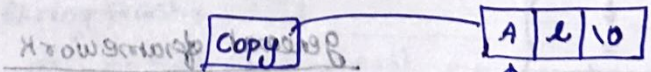
data of address of pointer = *(& pointer)

= pointer

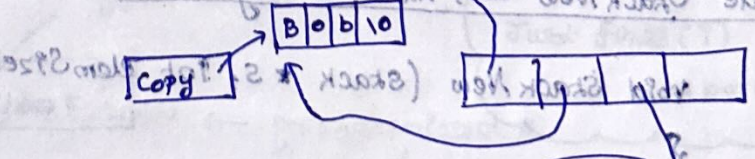
= (which has address!)

---

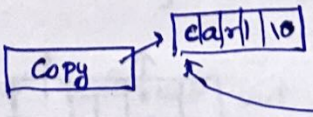
① 1st iteration

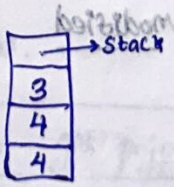


② 2nd iteration



③ 3rd iteration





Now: Stack Pop (give me them).

```
char *name;
for (int i=0; i<3; i++) {
    StackPop(&stringStack, &name);
    printf("%s\n", name);
    free(name);
}
StackDispose(&stringStack);
```

now write in name address!

Someone process!

name - no longer needed!

---

what happens? : String Stack Instead of & stringStack

```
memcpy(stringStack->elems, ____, ____);
```

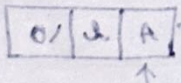
* Instead of copying to address] Core dumped!
* I am copying to value - dereferencing

---

* when free - Be sure → No longer needed!
* like we are free-ing name each time → StackDispose need a way to despose!

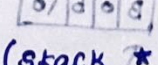
---

* Challenges: don't know pointers - Just 4 bytes!
* maybe: Struck with 3 pointers, ...
* we know how many elements: loglength

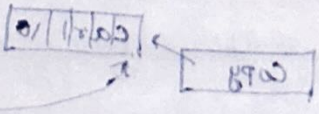


general framework

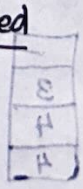
upgrade StackNew → take 3 arguments:



```
void StackNew(stack *s, int elemSize, void (*freefn)(void*))
{
    ____;
}
```

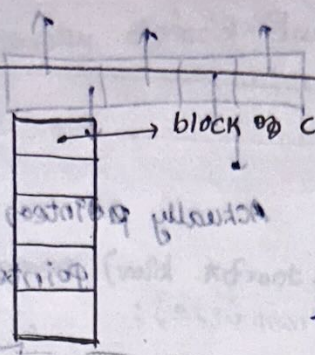


Modified



```
typedef struct {
    void *elems;
    int elemSize;
    int loglength;
```

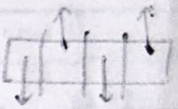
```
int allocLength;
void (*freeFn)(void*);
};
```



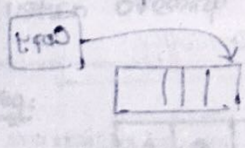
Note: int, float, double: Just (s→elem) free enough → not need to free element by element.

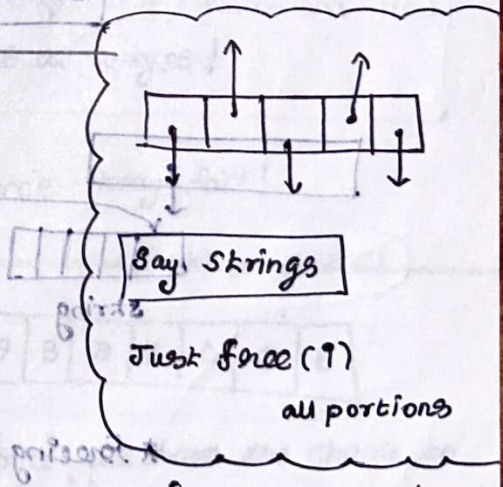
* No freeing needed (base types): int, f, doub → NULL (so understand: no need)
* char*, pointers to struct, struct. → place meaningful function to free!

```
void StackDispose(Stack *s)
{
    if (s->freeFn != NULL) {
        for (int i = 0; i < s->logLength; i++) {
            s->freeFn((char*)s->elems + i * s->elemSize);
        }
    }
    free(s->elems);
}
```



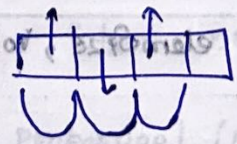
```
(say:) Stack stringStack;
       StackNew(&stringStack,
                sizeof(char*),
                StringFree);
```





knowing char*

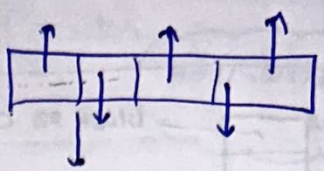
my free function i/p



I know: dereference them – get pointer!

```
void StringFree(void *elem)
{
    free(*(char **)elem);
}
```

Why:



Actually pointers having pointers pointing chars

char **

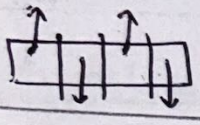
Forget: (char *) elem;

char * → means I'm passing that elem (pointer eg pointer alone without deref)

deref to get to char *

free knows → it's not a dynamic memory (pointer)

Deallocation?



∴ So they (strings) won't be freed

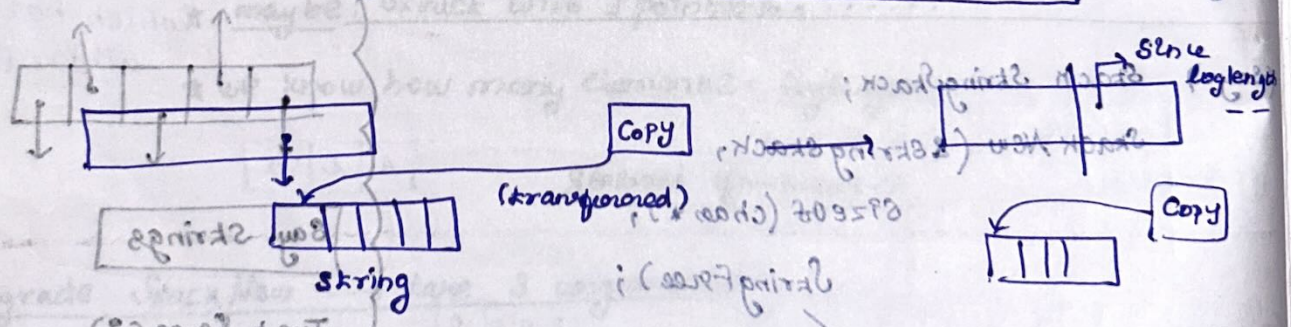
I need to free strings! (donate to heap)

Doing: free the elems from strings; strings actually there!

---

why not free? when pop → memcpy(dest, source, —)

Because: memcpy → writing addresses



* freeing string → checks (Copy) since it's pointing

* No use of poping at all.

```
void StackNew(stack *s, int elemSize, void (*freeFn)(void *))
{
    ...
    s->freeFn = freeFn;
}
```

why not free → in main → after strdup.

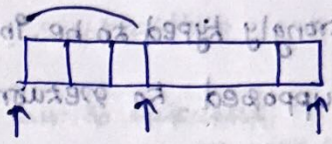
Reason: doing that - free string - But string still a part of stack

Actually freed during Pop / Dispose! (heap).

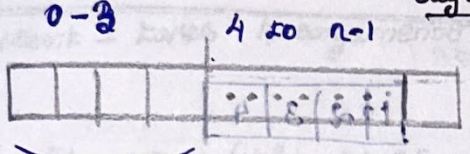
↳ By myself (client) → By our function! (Server)

Rotate [STL library]

```
void rotate (void *front, void *middle, void * end)
{
  // I want circular rotation
}
```



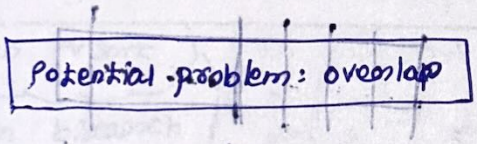
end - address of 1st byte - nothing to do with this array.



say - 4 shift

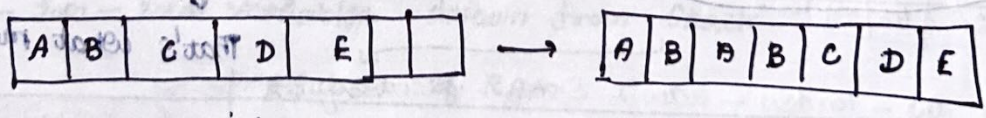
copy → 4 to n-1 to 0 to (n-1)-4

then (n-1)-4 to (n-1)



* memcpy → brute force which does 4 bytes at a type!
* Assumes: overlapping → No!
* when overlap → memcpy & brute force may fail!

eg:



Problem: we have changed C, D, E → before we have the chance to copy it!

client - needs to take care of it

* or: use less efficient prototype! (which does overlap checking)!

memmove → may be (name) inspite of relocating - Just moving!

```
void rotate (void *front, void *middle, void *end)
{
    int frontSize = (char *) middle - (char *) front;
    int backSize = (char *) end - (char *) middle;
    char buffer [frontSize];
    memcpy (buffer, front, frontSize);
    memmove (front, middle, backSize);
    memcpy ((char *) end - frontSize, buffer, frontSize);
}
```

→ Same size as first few elements

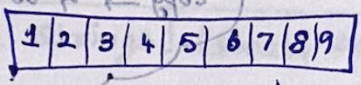
Strongly typed to be int sizes → do pointer subtraction b/w two (int*)

Supposed to return: no. of ints that could fit b/w them!

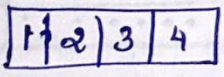
"Consistent with pointer arithmetic"

(char *) − (char *) ∴ 1 byte → Same as arg. subtraction

Step: 1

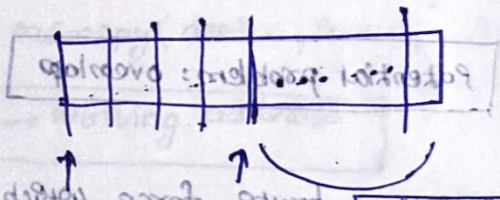


4 − shift!



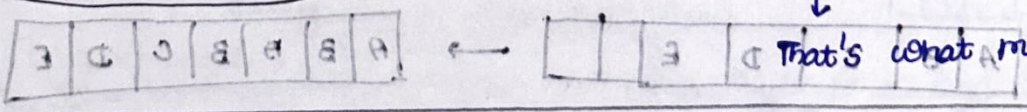
"Buffer"

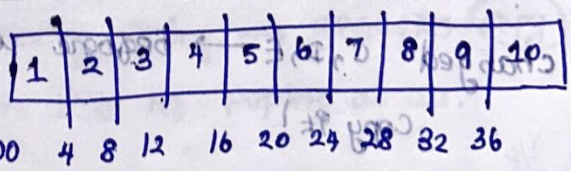
Call memmove: when you have to overlap



Do: overlap check
Then memcopy

memcpy → efficient!





Shift: 4 bytes: elements

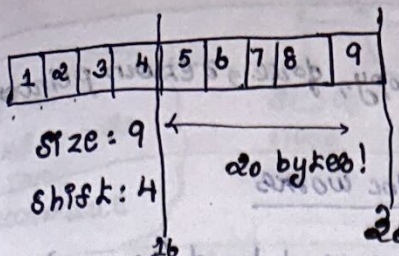
front: 0000
middle: 0016
end: 0040

(or)

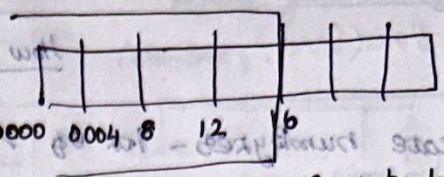
Extra Job!

only going to use to subtract!

Foresh



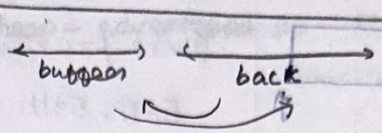
I dont want elemsize: byte shifting copying.



16 bytes – Copy to buffer!

---

when I get 3 pointers:



I can Compute

Interchange 20 bytes by pointer arithmetic: Subtraction

Here: No pointer arithmetic: get elemSize! → To calculate bytes need to interchange.

---

quicksort – array, elemSize, no. of elements, comparison func.

→ qsort – takes 4 arguments! – fast sorting algorithm

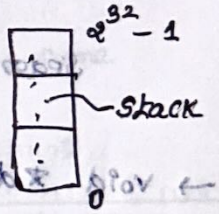
```
void qsort (void *base, int size, int elemSize,
            int (*cmpfn)(void *, void *));
```

>man qsort

>man bsearch

> memcpy, malloc, realloc, memmove

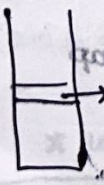
32 bits



Pointers: $2^{32}-1$ possible (Say!)

---

memory – for – local variables drawn from stack

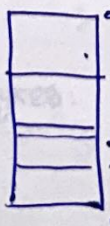


* Subset of RAM
* helper function: (main) – variables – helper variables: actives – after that destroyed come to main → enabled.

---

why: Stack: most recently called – invited to return (So the Stack: name)

*



Addresses managed by hardware

Assembly code

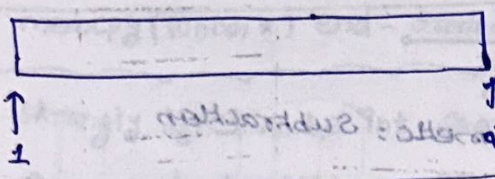
*heap - managed by software! (malloc, realloc, calloc)

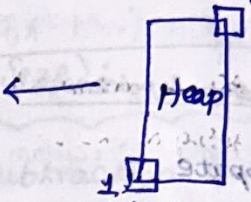
realloc → extend (available)
↳ reallocate (By search new, allocate, copy, free, return pointer)

How malloc, realloc, calloc works

* Allocate numBytes - lot of scenes happening behind hood (new...)

Heap





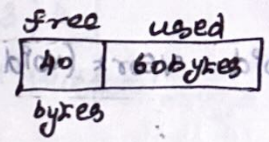
drawn as one large rectangle!

```
void *a = malloc(40);
void *b = malloc(60);
```

Search from beginning - search for 40 [count, enough - return]!
first open block - satisfy request

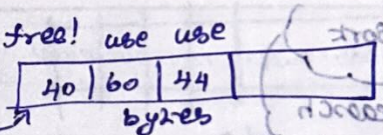
→ free(a); → 40 bytes → open for use!

→ void *c = malloc(44); // This



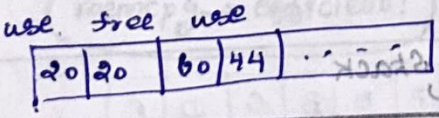
can't enough: 40

Search for 44



→ void *d = malloc(20);

Note: Some implementation start from where ended!

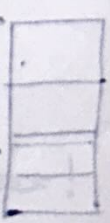


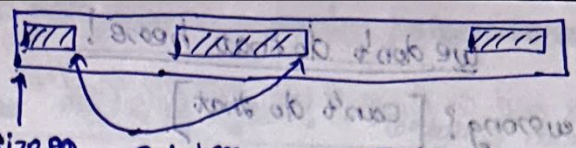
* OS - loader - admit space - memory address
* everything else in RAM - managed by our variable heap manager/memory allocator

→ realloc (60, 90) → space available - just extend

→ Memory allocator keeps
free notes - which are free!

* keeps 1st address
* see more clearly





freelist which node can accomodate

Size of node

Pointer to next one → free

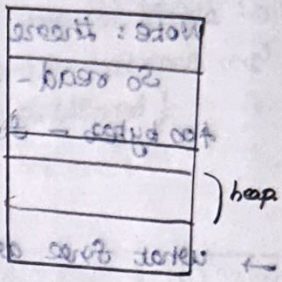
heap manager - need to do it more generically (no-int, ...)

Lecture - 8

(heap)

* managed - by Software include in language library - everytime - linked!

* lowest & highest address of heap - advertised to the lib. func going to use.



malloc, free, realloc — managed by C-Software

A person can use whatever (heuristic - learn something) - to make it correctly, quickly, efficiently as possible.

int *arr = malloc (40 * sizeof (int))

[Strongly type]

more than 100 bytes booked:

passed.

when a pointer go - to (free / realloc) → it assumes → that pointer is the one previously handed by either a call to malloc / realloc → some legitimate pointer.

→ way: memory allocated — more that what we ask!

→ why? * usually 4 bytes / 8 bytes extra → pointer!

4 bytes

(how big the node is)

* we always get pointer - 4/8 bytes - offset to the array.

* If I already given the pointer - I have the note - how big it is.

* use info: free the memory!

---

why don't work:

int *arr = malloc (100 * sizeof (int));

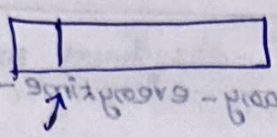
404/408 bytes: (realized: 60 int enough!)

free (arr + 60); → I want to clean 40 extra int spaces!

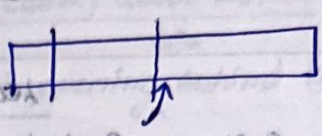
We don't do that here!

free (arr+60); → wrong? [Can't do that]

Say: free (arr);



Note: there
So read - 100 elements
400 bytes - free it!



Read previous 4/8 bytes — assume that
58th, 59th index as note!
'free using that info'

→ what free assumes: You are C++/C programmer — No error check!
→ To be efficient. — No error check.

Say: 25000 in 58/59 index

25000 bytes freed → Catastrophe.

---

* int array[100]; → we are defining it statically. — not in heap
* free (array); → go to the prev 4 bytes — do free that many bytes

Note: no error checking — to check whether array is in heap memory

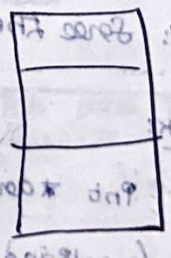
(Joined)
Then incorporated — as - free list.

Free list — Collection of free memory - Future use!

* Fastest: No error checking!
* One implementation: Keep track of all (void *) → handed back
  Quick check — Is it a member → handed over!
  If error detected: Ignored! (later: going to cause problems)

Implementation

→ 160 → not a power of 2



→ Some implementation
→ Always gives out: 64 bytes/
8 bytes long.

perfectly size: takes time → So just gives 64/8...
that order.

★ Packet – get your package! [Size bucket : 2^3, 2^6, 2^9]

160 → Allocates 192 / 256 bytes! → Certainly more memory to satisfy our need.

Implementation – may differ – Gcc / Codewarrior (vs) MCode on mac.

Sometimes: overrun – doesn't Causes array error!

$i < 10 \longrightarrow i \leq 10$

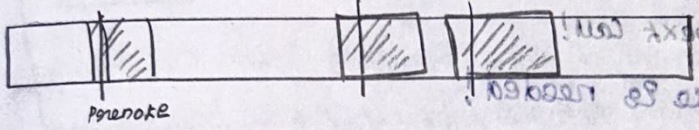
(may be due to implementation)

2 diff engineers

Can't assume: memory (as imple. varies)

→ array – 1 (Say: we known where it is – but byte malloc, free relying Can be modified → say all the info: Risk! → So access must be declined!

array [-1]



Some implementation:

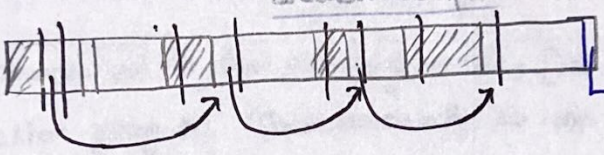
how big/

node after it is free/not

So optimize with trade off memory

★ Blank nodes: quickly hop – without scanning each one!

★

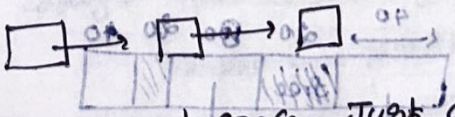


→ NULL (or) Cycle back!

Pointer to next unused heap! → So scanning is reduced!

★ every time malloc/calloc/realloc → traverse this linked list ←

★ It's like: free blocks: Connected as linked list ←



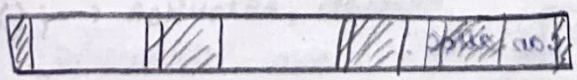
go to 1st: enough space: Just allocate

else: go to 2nd: using pointer held by 1.

(or) 'until big enough node'!

This kind: no need for too much scan: → Counter each time!

Another implementation – not common



Entire heap by exhaustive search

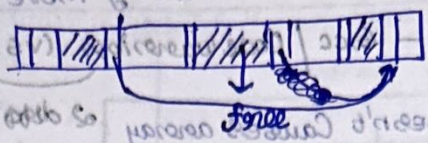
↳ 'Best fit'

So only minimum space required!

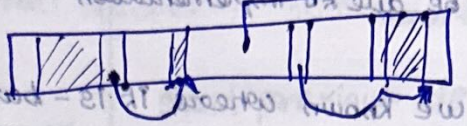
Worse fit strategy: search biggest one – idea: part left by it will still fair enough – for further use!

* Why? Normal book: 4/8 bytes - useless for others!

→ next call continue from where it left off!



So updated ≥ ? → Force (info remains - not cleared - But for hire).



Just rewrite 'don't clean' → time consuming

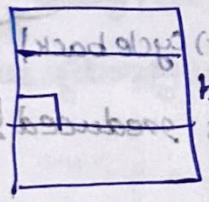
→ Rewrite info - why delete? (client supposed to use again!

Just updated! (That time is for hire)!

Some cases: Just record - doesn't free until next malloc call. - only commit to free after next call!

Some cases: until only when space is needed!

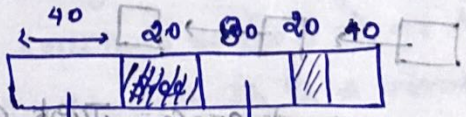
heap-structure



→ Address of entire free list

→ First four bytes: NULL (size of heap)

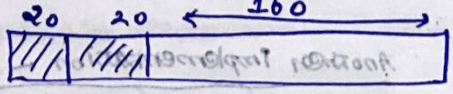
→



Best fit Worst fit

malloc (40)

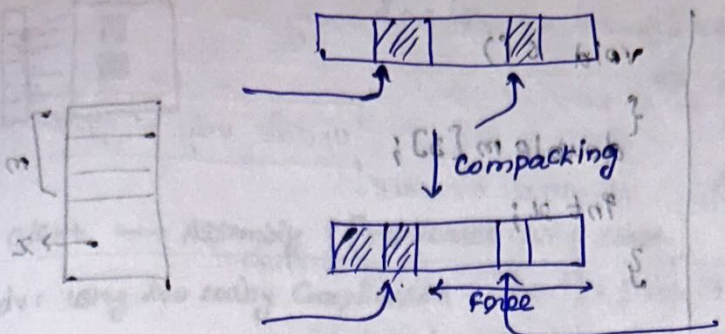
* malloc(100) → Not enough continuous space



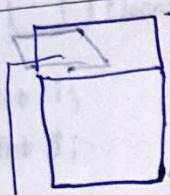
Now I can alloc.

* Compacting - allocate close as possible!

* very likely: Problem: move the data!



* problem: client is still pointing the same space. (whereas is the data)
* client doesn't know moved.
* macintosh used - 1990's.



Some OS - uses handles (pointers two hops away)

maintain: list of single pointers. (handout - free memory void * to client).

maintain table → Adv: owned by heap manager

Can update! - without affecting void**

macos 7.6
1994-95

---

problem: Compacted in low priority thread, [no space: Can't actually have heap compaction going on simultaneously to an actual access request. while accessing it.

get memory - managed aggressively

void **handle = New Handle (40);

problem: Can't read while moving!

Safety: HandleLock (handle); → so no moving-locked.

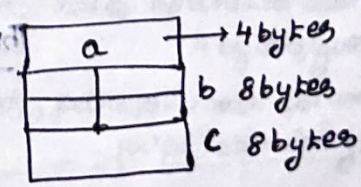
HandleUnlock (handle); → unlocked!

---

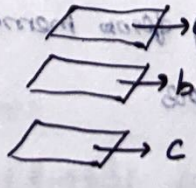
void ()
{
  int a;
  short b[4];
  double c;
} disabled. (Stack)

B(); → Activated, disabled

C(); → Activated, disabled.



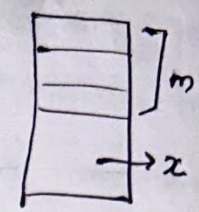
Activation record/ Stack frame.



'plates'
→ Has track of records
→ incremental

```
void B()
{
  int x;
  int *y;
  char *z[2];
  c();
}
```

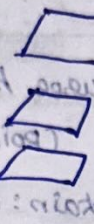
```
void c()
{
  double m[3];
  int x;
}
```

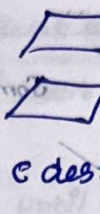


a → b → c

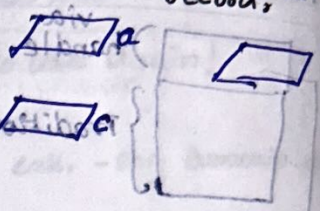
a → c

→ Pointers to layers/frame → points to most recently called functions activation record,





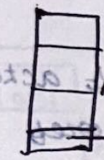
c destroyed



→ when returns: data actually there: we don't have access anymore

→ B doesn't even know C was called. (don't know how to access C's data)

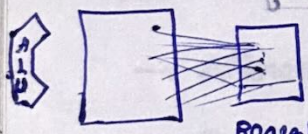
→



* Code Segment - Assembly code from Compiled C, C++

*

ALU



RAM

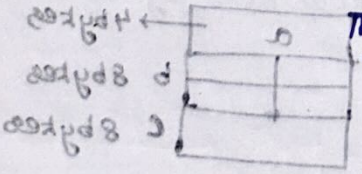
Registers (Fastest) - 32

Computer processor

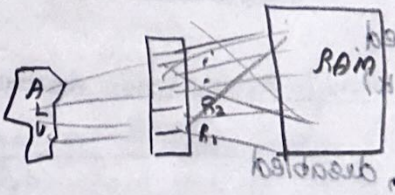
Registers - general purpose * 1 byte figure.

Registers - in touch with RAM

ALU - do math +, -, &&, mod, /, //...



This imple: ALU touch with Registers - Not directly to RAM



* Take something from memory

* Load in Registers

* Do operations

RAM - expensive/ slow!

"Load - Store" - Registers - Cost/Speed eff.

g += 1;

① Load f, g in Registers

② $R_1 = 7$

$R_2 = 10$

$R_1 + R_2$ → Then Store! (Flush out).

How c/c++ → Assembly ? [replicates c/c++ state..]

Disadv: Very too many Complicated - matrix - circuit b/w Reg & RAM.

else: Slow, more operations! ← Best one! (trade off: Complexity).

* more Complicated hardware Implementation: Clock Cycle speed goes up

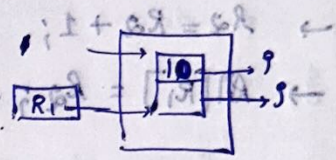
Lecture-9 - Assembly code!


```
int i;
int j;
f = 10;
j = f + 7;
j++;
```

Assembly code

* Registers (special address)

* To get 10:



→ M[R_1+4] = 10; //Store operation!

Base addr, offset!

M → Name of ALU of RAM.

→ Fetch, Add, Flush out!

→ R_2 = M[R_1+4]; // load f

→ R_3 = R_2 + 7; // add 7

→ M[R_1] = R_3 // flush out

// Load operation.

// ALU operation

$R_1, \dots R_n$ → general purpose registers.

// R_1 has base address

R_2 = M[R_1];
R_2 = R_2 + 1;
M[R_1] = R_2;
} 3 line [j = j + 1;]

// Reuse registers;

// overwrite allowed;

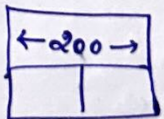
// OS itself optimized deal with 4 byte figures

// int, pointer - most commonly used atomic types

"Sequential - must"

```
int f;
short s1;
short s2;
```

f = 200;
s_1 = f;
s_2 = s_1 + 0;



→ R1 (Recent one address)

M[R_1+4] = 200;

M[R_1+2] = M[R_1+4];

can't do Load & Store at a time.

R_1+4 → int f.

M[R1+4] → load from memory.

→ R2 = M[R1+4];

→ M[R1+2] = R2; → won't do what we want.

update only 2 bytes

update 4 bytes from this memory,

.2 → make it 2 byte!

→ M[R1+2] = .2 R2; (lower half)

→ R2 = M[R1+2]; // get S1

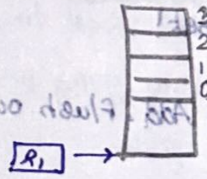
→ R2 = R2 + 1;

→ M[R1] = R2; → 4 bytes (By default) → wrong

M[R1] = .2 R2; [2 byte].

```
int array[4];
int i;
for (i=0; i<4; i++){
    array[i] = 0;
}
i = -1;
```

→ Relational operators!



Assembly

M[R1] = 0; // Initialize. → PC+4

R2 = M[R1]; // load → PC+4

BGE R2, 4, ____ → PC+8

each inst. 4 bytes?

→ PC + ____ (Target)

B__ → Branching

R2, 4, ____ Target

G,E → Greater than / Equal to

BEq → ==

BNE → !=

BLT → <

BLE → <=

BGT → >

BGE → >=

PC → program Counter (stores address of current instruction)

$M[R_1] = 0;$

$R_2 = M[R_1];$ ⟶ PC + 4

BGE $R_2, 4, PC + 40$ (after 40 (10 lines)) – false.

$R_3 = M[R_1];$

$R_4 = R_3 * 4;$ ⟶ // Assume: int. (not the case)

$R_5 = R_1 + 4;$ ⟶ // offset [array] ("distance: got from 9*4)

$R_6 = R_4 + R_5;$ ⟶ array+0, array+4, array+8 ...

$M[R_6] = 0;$

$R_2 = M[R_1];$ ⟶ // load 9

$R_2 = R_2 + 1;$ // 9++;

$M[R_1] = R_2;$

JMP $PC - 40;$ Jump by 10 instructions: 10×4 = 40

(after loop)

$R_2 = M[R_1];$ ⟶ PC + 40 // 9--;

$R_2 = R_2 - 1;$

$M[R_1] = R_2;$

(explicitly writing)

C – Just foo(...) { }

By default: 4 Byte figure

---

why 32 bit (4 byte enough)?

Assembly: subdivided in to subpackets

$R_1 = M[R_2 + 4];$

$R_1 = 1000;$

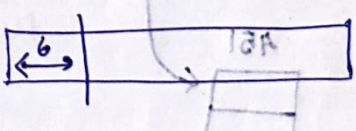
$R_3 = R_6 * R_{10};$

$M[R_1 - 20] = R_{19};$

Instructions (load, assign, ALU, Flush out)

In this 4 bytes (Say 59 Codewords)

Identify which code? word



Say: each having op code

$R_1 = M[R_2 + 4]$ ⟶ 00000 (say load)

Immediate add ⟶ 0001

Dest codes: dest op code!

$R_1 = M[R_2 + 4];$

4 bytes (Identify codeword) | 5 bytes | 5 bits

16 more bits ⟶ assigned operat.

5, 5, 16 ⟶ relevant when 000000 (valid)

$R_1 = 1000;$ ⟶ 000001 ⟶ all the remaining bits → can be for int/value.

For each opCode: Set some bit pattern

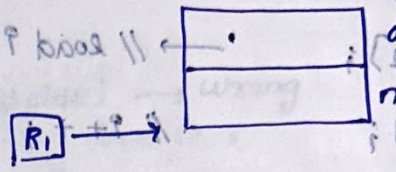
All opCodes: Same size.

Say: 000 (opcode), 001001

★ MIPS: 000 (Common) ↳ "don't interpret as a bit"

Pointers & Casting

```
Struct fraction{
    int num;
    int dnum;
};
```




```
Struct fraction pf;
pf.num = 22;   ⟶ M[R1] = 22;
pf.dnum = 7;   ⟶ M[R1+4] = 7;
```

"Forgotten about Structs"

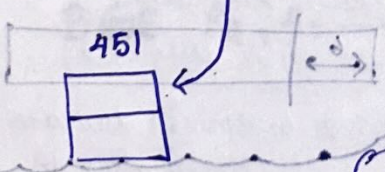
(&pf.dnum) ((Struct fraction *) &pf.dnum) → dnum = 451;

★ pf.num → base address!

★ So → (&pf.dnum) → dnum access top element — "Some random 4 bytes"

Above pf (Struct) = not owned by struct

"ghost" (we know doesn't exist) — But address there



M[R1+8] = 451;

(Struct fraction *)

(&pf.dnum) → dnum;

↓ 4 bytes offset

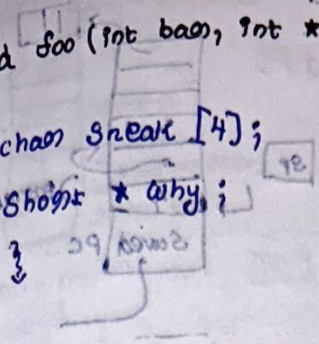
↓ (4 beyond already offset)! 8 bytes offset.

Rule: C → asc → no code instruction does casting!

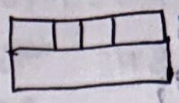
"Just Compiler things → cast → permission slip"

Lecture - 10

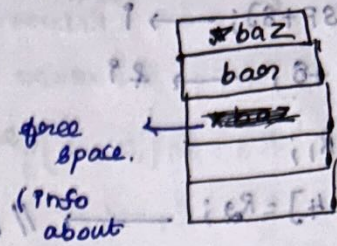
```
void foo (int baar, int * baz)
{ char sneak [4];
  short * why;
}
```



sneak [4]

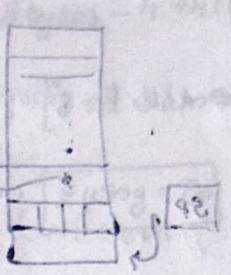


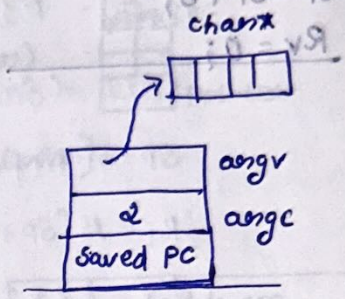
high to low → '0' th addresses below.



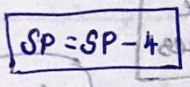
free space.

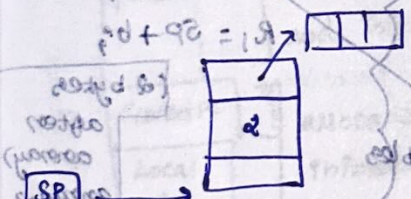
```
int main (int aargc, char **aargv)
{ int q = 4;
  foo (q, &q);
  return 0;
}
```





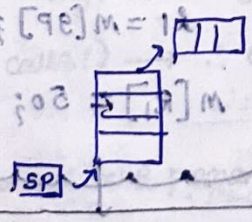
* C function: makes space upon local variables: activation record.
* with a has activation record → Complete full activation record.

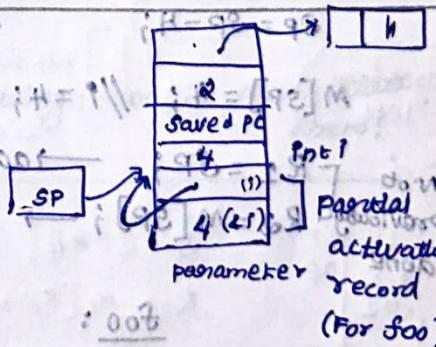




dedicated registers
SP - stack pointer
(always the thing truly point to the lowest address in the stack

q = 4 → denotes this sp by 4 more bytes



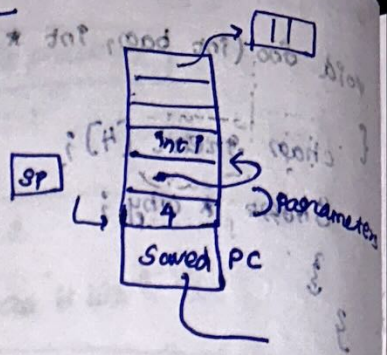


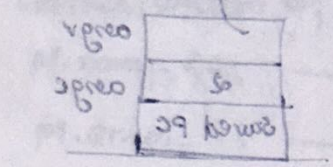

```
SP = SP - 4;
M[SP] = 4;
SP = SP - 8;
R1 = M[SP + 8];
R2 = SP + 8;
M[SP] = R1;
M[SP + 4] = R2;
```

→ R1 has value
R2 has address

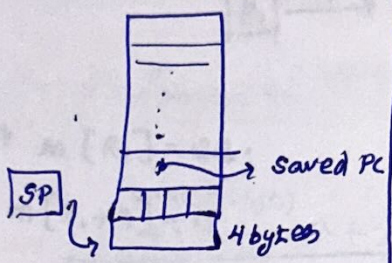
Transfer Control to foo

```
SP = SP - 4;
M[SP] = 4;
SP = SP - 8;
R1 = M[SP+8];   → f
R2 = SP + 8;    → &f
M[SP] = R1;
M[SP+4] = R2;   → // get save pc details. [Automatically when call <foo> is seen]
CALL <foo>.
SP = SP + 8;
Rv = 0;
```





```
foo: SP = SP - 8;
```

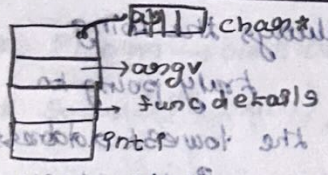



```
void foo (int boon, int *boo)
{
  char sneak[4];
  short *why;
  why = (short *) (sneak+2)
  *why = 50
}
```

PC, SP ← Complete activation record!

Rv - communicate return values

dedicated registers.



call foo – 2 parameters

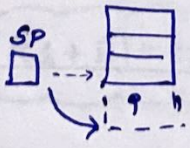
step: set 2 param

first f=4,

foo(f, &f);

So for f

```
SP = SP - 4;
M[SP] = 4;   // f = 4;
```



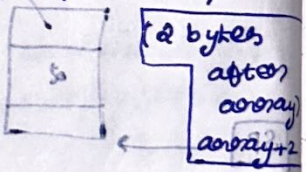

```
R1 = SP;    → address of f
R2 = M[SP]; → value of f
```

Not previously done

foo:

only sees function – it has parameters as foo call then only parameters created.

```
foo: SP = SP - 8;
     R1 = SP + 6;
```




```
M[SP] = R1; [why]
R1 = M[SP];
M[R1] = 50;
```

Promote between all started.

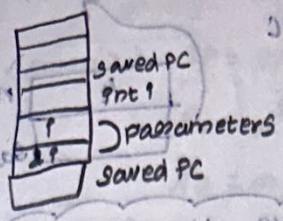
```
SP = SP + 8;   (saved pc of foo)
RET;
```

→ call blocks after saved PC vanishes!

Discard 2 parameters (deallocate)!

done!

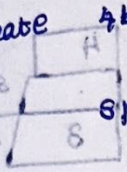
Function call seen: Create parameters; PC (saved PC)




```
SP = SP - 8;   (Foo parameters)
R1 = M[SP+8];  // get i
R2 = SP+8;     // &i
M[SP] = R1;
M[SP+4] = R2;
```

∴ All done → call <foo>

Create 4 byte char array, 1 short pointer




```
SP = SP - 8;   // create 8 bytes - 4 + 4 (pointer)
```

why = (short *) (&weak+2); [3rd element of &weak]

```
R1 = SP+6;
R1 = M[SP];
M[R1] = 50;
```

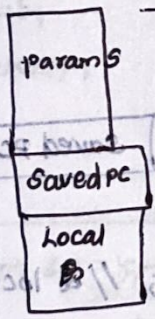
→ M[SP] = R1; → *why = 50

write address of &weak+2 in why

*why = 50

Load address

M[address] = 50



alloced and initied by callee

allocated by function call foo!

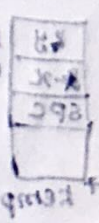
initiated

why? main get space for all variable? (Instead main take care of calls?) → ∴ only main knows how to put meaningful info!

callee has no idea how many local var. involved

* callee sets parameter etc.. in first half.

* doesn't mean count of local var: unknown: meaning: how to manipulate them!

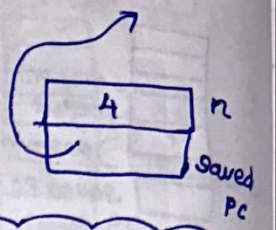


<call> ; RET

Jump statements.

```
int fact (int n)
{ if (n==0)
    return 1;
  return n * fact (n-1); }
```

* Compilers: look for all local variables - compute size - allocate!



<fact> : R1 = M[SP+4]; // get n → above Saved PC

PC BNE R1,0, PC+12 (3 statements)

PC+4 RV = 1; (Return value pointer)!

PC+8 RET; // where it's been called.

PC+12 R1 = M[SP+4]; // load n

R1 = R1 - 1; // n-1

SP = SP-4; // for parameter.

M[SP] = R1;

CALL <fact>

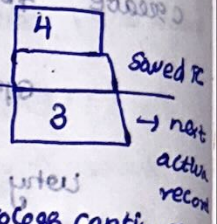
SP = SP+4; (gets rid of Saved PC)

R1 = M[SP+4];

RV = RV * R1; // n*fact(n-1)

RET;

BNE Not equal - have branch else don't



Process continues

R1 → n-1

RV → Return value from function!

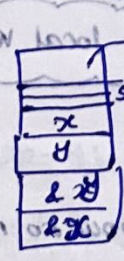
factorial trace! → PDF (See Stanford: Prog paradigms)

Lecture-11

```
void foo()
{
  int x;
  int y;
  x = 11;
  y = 17;
  swap(&x, &y);
}

void swap(int *ap, int *bp)
{
  int *temp = *ap;
  *ap = *bp;
  *bp = temp;
}
```

Stack pointer → Saved PC

main:
SP = SP-8; // 2 locals
M[SP+4] = 11;
M[SP] = 17;

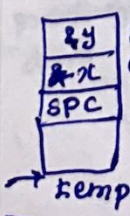


Right from left

param Partial activation record.

Parameter Creation:

R1 = SP; // &y
R2 = SP+4; // *x
SP = SP-8; // For parameter
M[SP] = R2;
M[SP+4] = R1;
CALL <SWAP>

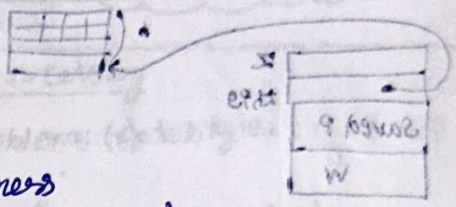


SP = SP-4; //temp
R1 = M[SP+8]; //&x
R2 = M[R1];
M[SP] = R2; //temp = &x
R1 = M[SP+12]; R2 = M[R1]; R3 = M[SP+8]; M[R2] = R2

SP = SP+8; deallocation
RET; return (right after start)

→ M[M[SP+8]]; → No architecture supports double reference

<swap>: SP = SP − 4;
R1 = M[SP+8];
R2 = M[R1];
M[SP] = R2;
R1 = M[SP+12];
R2 = M[R1];
R3 = M[SP+8];
M[R3] = R2;
R1 = M[SP];
R2 = M[SP+12];
M[R2] = R1;
SP = SP+4; → Saved PC
RET → Saved PC → explored → Return to call statement!

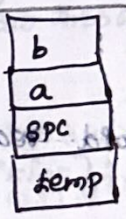


get address
M[address] = value!

```
void swap (int &a, int &b)
{
  int temp = a;
  a = b;
  b = temp;
}
```

```
int x = 17;
int y = x;
int &z = y;
```

```
int x;
int y;
x = 11;
y = 17;
swap(x, y);
```



values (some memory!)

→ int &z = y; → wrong!
→ C++ → reference & objects [references → less confusing pointers].
[objects → structs (methods declared inside)]

Structs & classes (C++) → only difference

default modifiers - classes – private
struct – public

★ struct has methods!

```
class binky {
  public:
    int dunky (int x, int y);
    char * minky (int * z);
    int w = *z;
    return blinky + dunky (winky, winky);
  private:
    int winky;
    char * blinky;
};
```

```
char blinky [4];
};
```