COS 217: Introduction to Programming Systems

Structures,
Command Line Arguments,
Dynamic Memory

PRINCETON UNIVERSITY

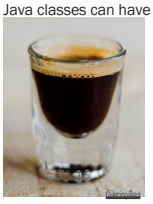
1

C STRUCTURES

2

{new state, updated line number}

- Java classes can have many fields



- How to get the equivalent in C?

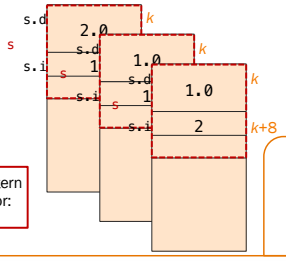
3

Add some structure to your program

```
struct S {
  double d;
  int i;
};

struct S s = {2.0, 1};
struct S* ps = &s;

s.d = s.i;
(*ps).i *= 2;
```



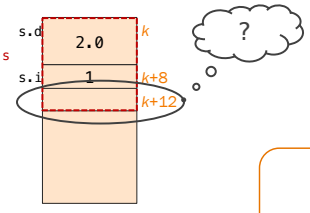
This is such a common pattern that it has its own operator:
`ps->i`

4

struct instruction

```
struct S {
  double d;
  int i;
};

struct S s = {2.0, 1};
```

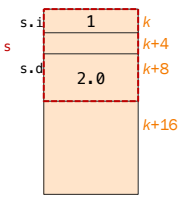


5

eventually I'll tire of visual puns (or not)

```
struct S {
  int i;
  double d;
};

struct S s = {1, 2.0};
```

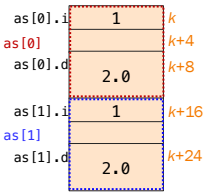


6

~~structstructstructstructstructstruct~~

```
struct S {
  int i;
  double d;
};

struct S as[2] =
  { {1, 2.0} };
as[1] = as[0];
```



7

7

struct construction, what's your function?

```
void printS(struct S s) {
  printf("%d %f\n", s.i, s.d); }

void swap1(struct S s) {
  int temp = s.d;
  s.d = s.i;
  s.i = temp; }

struct S swap2(struct S s) {
  int temp = s.d;
  s.d = s.i;
  s.i = temp;
  return s; }

void swap3(struct S* ps) {
  int temp = ps->d;
  ps->d = ps->i;
  ps->i = temp; }
```

```
int main(void) {
   struct S s = {1, 2.0};
   printS(s);
   swap1(s);
   printS(s);
   s = swap2(s);
   printS(s);
   swap3(&s);
   printS(s);
   return 0;
}

armlab01:~/Test$ ./sswap
1 2.000000
1 2.000000
2 1.000000
1 2.000000
```

8

8

Whose Rules Rule?

```
struct S {
  int arr[10];
};

void printS(struct S s) {
  int i;
  for(i = 0; i < 10; i++)
    printf("%d ", s.arr[i]);
  printf("\n");
}
```

```
int main(void) {
  struct S s = { {0,1,2,3,4,5} };
  struct S s2 = s;
  printS(s2);
  return 0;
}

armlab01:~/Test$ ./sa
0 1 2 3 4 5 0 0 0 0
```

How many int arrays are stored in memory?

A. 0: arrays in a struct aren't really arrays
B. 1: arrays are passed with a pointer
C. 2: structs are copied on assignment
D. 3: plus structs are passed by value
E. Arrays can't be fields of a structure.

The correct answer is D.

Passing, returning, or assigning a structure with an array field copies the array by value (a deep copy)!

9

9

COMMAND LINE ARGUMENTS

10

10

What's my name?

- String[] args was COS 126 day 1

- How to get the equivalent in C?

11

11

With sed s/s/v/ , natch.

```
int main(int argc, char* argv[])
{
   int i;

   /* Write the command-line argument count to stdout. */
   printf("argc:  %d\n", argc);

   /* Write the command-line arguments to stdout. */
   for (i = 0; i < argc; i++)
      printf("argv[%d]:  %s\n", i, argv[i]);

   return 0;
}
```

As parameters, these are identical:
char a[] and char* a
So it follows that, as parameters, these are, too:
char* argv[] and char** argv

12

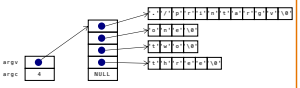
12

Elucidating Example: Explanatory Echo

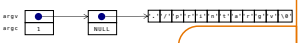
```
int main(int argc, char* argv[])
{
  int i;
  printf("argc:  %d\n", argc);

  for (i = 0; i < argc; i++)
    printf("argv[%d]:  %s\n", i,
                              argv[i]);
  return 0;
}
```

```
$ ./printargv one two three
```



```
$ ./printargv
```



13

13

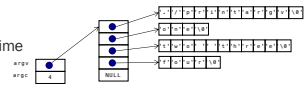
What's argc?

```
./printargv one "two  three" four
```

A. 3
B. 4
C. 5
D. Syntax error at runtime

B:

```
$ ./printargv one "two  three" four
```



14

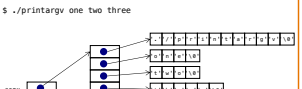
14

A2-inspired: rewrite everything in arrays to use pointers

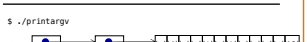
```
int main(int argc, char* argv[])
{
  char** ppc = argv;
  printf("argc:  %d\n", argc);

  while(*ppc != NULL)
    printf("argv[%d]:  %s\n",
                ppc-argv, *ppc++);
  return 0;
}
```

```
$ ./printargv one two three
```



```
$ ./printargv
```



15

15

Kicking the extra point?

```
int main(int argc, char* argv[])
{
  char** ppc = argv;
  int i = 0;
  printf("argc:  %d\n", argc);

  while(*ppc != NULL)
    printf("argv[%d]:  %s\n", i++,
                          *ppc++);
  return 0;
}
```

?

```
int main(int argc, char* argv[])
{
  char* pc = *argv;
  int i = 0;
  printf("argc:  %d\n", argc);

  while(pc != NULL)
    printf("argv[%d]:  %s\n", i++,
                          pc++);
  return 0;
}
```

A. Yes! This works and is clearer.
B. Maybe. This works but is less clear.
C. No! This is incorrect!
D. No! This doesn't even compile!

C:

```
argc: 1
argv[0]: ./pcla-wrong
argv[1]: /pcla-wrong
argv[2]: pcla-wrong
argv[3]: cla-wrong
...
```

16

16

mainly nonsense

```
int main(int argc, char** argv) {
int retVal;
if(argc == 0) {
  return 0;
} else {
  retVal = main(argc-1, argv+1);
  printf("%d: %s\t", argc-1, argv[0]);
  return retVal;
}
}
```

The correct answer is B:

```
armlab01:~/Test$./recur-r a b c; echo
0: c 1: b 2: a 3: ./recur-r
```

C is only the case at the start of execution, and does not hold if the program changes `argc`.

D would be the behavior with `exit(retVal);` instead of `return retVal;`

What does this program do?

A. prints arguments
B. prints arguments in reverse order
C. recurs infinitely: `argc` is always ≥ 1
D. prints only the last argument: `return` from `main` exits the program

18

18

DYNAMIC MEMORY

19

19

Why, though?

- Thus far, all memory that we have used has had to be known at compile time.
- This is not feasible for realistic workloads; many times memory needs are dependent on runtime state
  - User input
  - Reading from a resource (file, network, etc.)
  - …

```
How many records are being entered?
```

20

20

Memory Allocation at Runtime

Thus far we have seen 3 memory sections:

- Stack
  - Function parameters and local variables
- Text
  - Program machine language code
- RODATA
  - Read-only data, e.g. string literals

Now: "Heap"

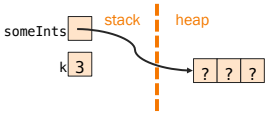
21

21

Your New Friends: malloc

```
int k;
int* someInts;
printf("How many ints?");
scanf("%d", &k);
someInts =
   malloc(k * sizeof(int));
```

```
int k;
int* someInts;
printf("How many ints?");
scanf("%d", &k);
someInts =
   calloc(k, sizeof(int));
```



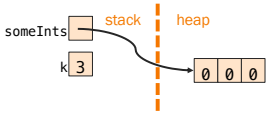
22

22

Your New Friends: calloc

```
int k;
int* someInts;
printf("How many ints?");
scanf("%d", &k);
someInts =
   malloc(k * sizeof(int));
```

```
int k;
int* someInts;
printf("How many ints?");
scanf("%d", &k);
someInts =
   calloc(k, sizeof(int));
```



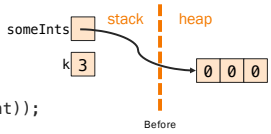
23

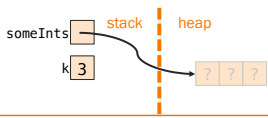
23

Your New Friends: free

```
int k;
int* someInts, *moreInts;
printf("How many ints?");
scanf("%d", &k);
someInts = calloc(k, sizeof(int));

free(someInts);
```





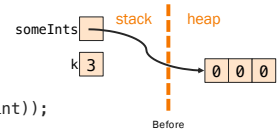
24

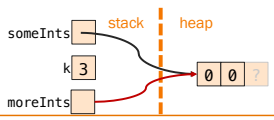
24

Your New Friends: realloc

```
int k;
int* someInts, *moreInts;
printf("How many ints?");
scanf("%d", &k);
someInts = calloc(k, sizeof(int));

moreInts = realloc(someInts, (k-1)*sizeof(int));
```



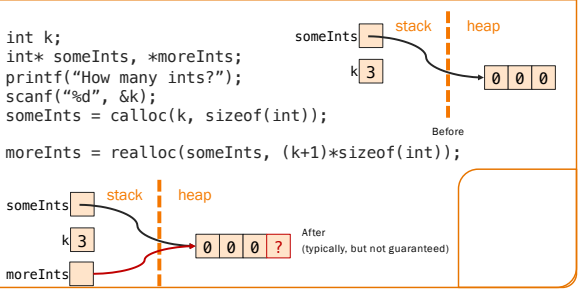


25

25

Your New Friends: realloc

```
int k;
int* someInts, *moreInts;
printf("How many ints?");
scanf("%d", &k);
someInts = calloc(k, sizeof(int));
moreInts = realloc(someInts, (k+1)*sizeof(int));
```

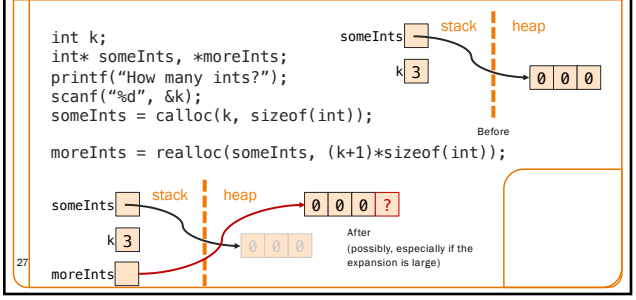


26

26

Your New Friends: realloc

```
int k;
int* someInts, *moreInts;
printf("How many ints?");
scanf("%d", &k);
someInts = calloc(k, sizeof(int));
moreInts = realloc(someInts, (k+1)*sizeof(int));
```

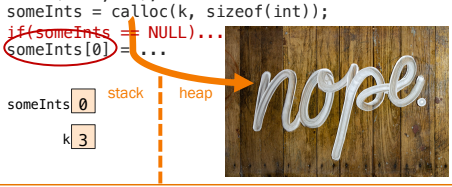


27

27

What could go wrong (malloc, calloc)?

```
int k;
int* someInts, *moreInts;
printf("How many ints?");
scanf("%d", &k);
someInts = calloc(k, sizeof(int));
if(someInts == NULL)...
someInts[0] = ...
```

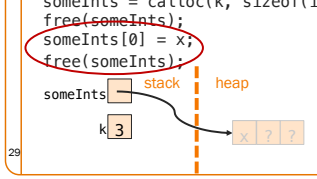


28

28

What could go wrong (free)?

```
int k;
int* someInts, *moreInts;
printf("How many ints?");
scanf("%d", &k);
someInts = calloc(k, sizeof(int));
free(someInts);
someInts[0] = x;
free(someInts);
```

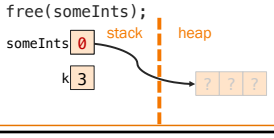


29

29

It's still a bug! (But now you'll find it!)

```
int k;
int* someInts, *moreInts;
printf("How many ints?");
scanf("%d", &k);
someInts = calloc(k, sizeof(int));
free(someInts);
someInts[0] = x;
free(someInts);
```

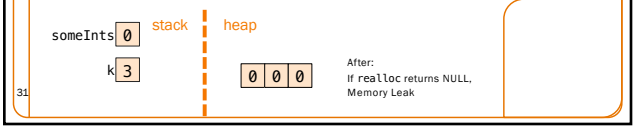


30

30

What could go wrong: realloc

```
int k;
int* someInts, *moreInts;
printf("How many ints?");
scanf("%d", &k);
someInts = calloc(k, sizeof(int));
someInts =
   realloc(someInts, (k+1)*sizeof(int));
```

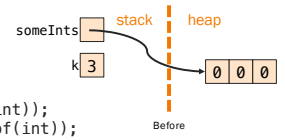


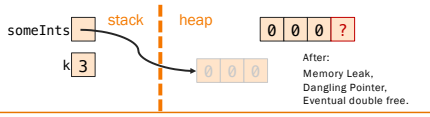
31

31

What could go even worse: realloc

```
int k;
int* someInts, *moreInts;
printf("How many ints?");
scanf("%d", &k);
someInts = calloc(k, sizeof(int));
realloc(someInts, (k+1)*sizeof(int));
```





After:
Memory Leak,
Dangling Pointer,
Eventual double free.

32

32

Catch the Most Common Bug

```
newCopy = malloc(strlen(oldCopy));
strcpy(newCopy, oldCopy);
```

Does this work?

A. Totally! (Wait, what's the title of this slide again?)

B. Nope! The bug is …

B:

This allocates 1 too few bytes for `newCopy`, because `strlen` doesn't count the trailing `'\0'`.

33

33

Save a line?

```
newCopy = strcpy(malloc(strlen(oldCopy)+1), oldCopy);
```

Does this work?

A. So *that's* why `strcpy` returns the destination! Sure!

B. Eh, okay, but this is less clear.

C. Nope!

C:

If `malloc` returns NULL, this fails the precondition for `strcpy`

34

34