CSE 374 Lecture 11

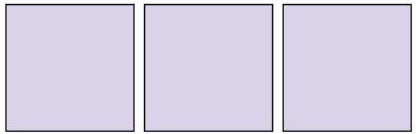
Memory Allocation & De-allocation

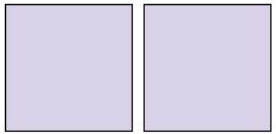
Puzzle: What prints?

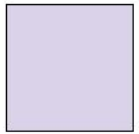
```
#include <stdio.h>
void mystery(char *a,int *b,
  int c) {
  int *d = b - 1;
  c = *b + c;
  *b = c - *d;
  *d = *b - *d;
  a[2] = a[b - d];
}
```

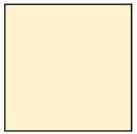
```
int main(int argc, char **argv) {
  char ant[4] = "bed";
  int x[2];
  *x = 6;
  x[1] = 7;
  int y = 4;
  int *z = &y;
  *z = *x;
  printf("%d %d %d %s\n", *x, \
         x[1], y, ant);
  mystery(ant, x + 1, y);
  printf("%d %d %d %s\n", *x, \
          x[1], y, ant);
}
```

Solve Mystery

ant 

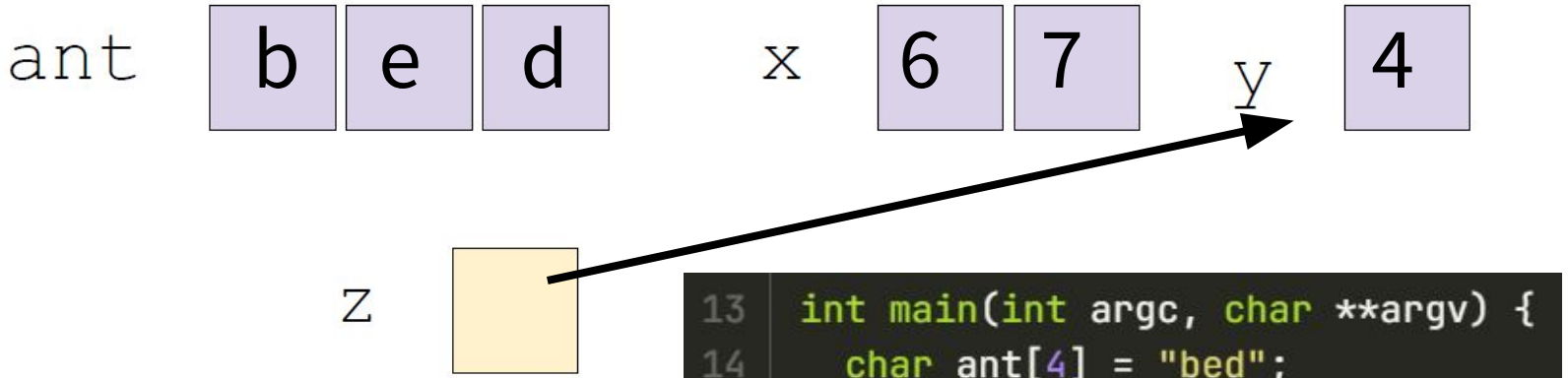
x 

y 

z 

```
13  int main(int argc, char **argv) {
14    char ant[4] = "bed";
15    int x[2];
16    *x = 6;
17    x[1] = 7;
18    int y = 4;
19    int *z = &y;
20    *z = *x;
```

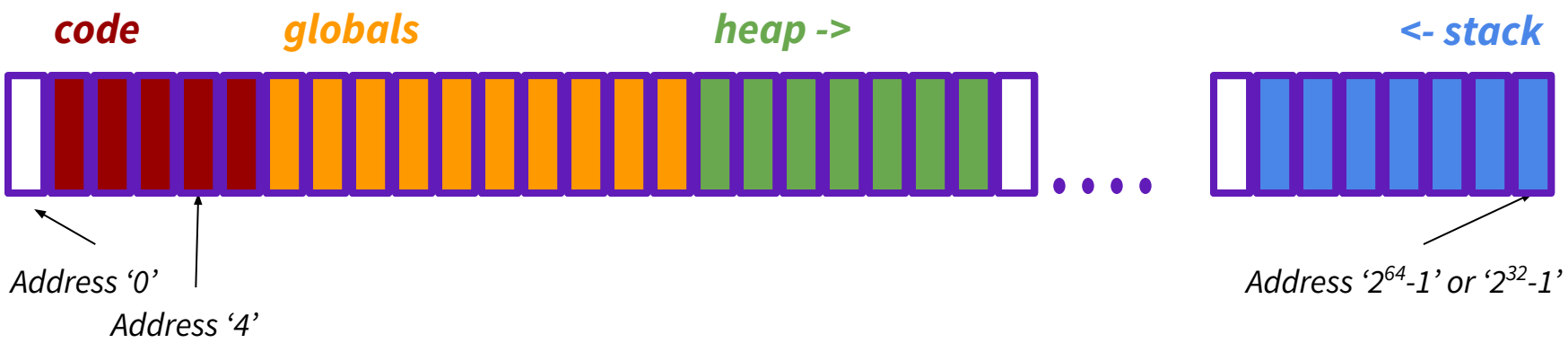
Solve Mystery



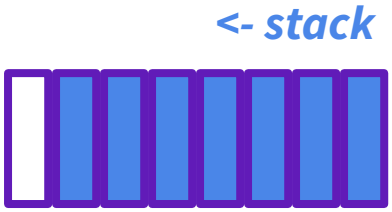

```
13  int main(int argc, char **argv) {
14    char ant[4] = "bed";
15    int x[2];
16    *x = 6;
17    x[1] = 7;
18    int y = 4;
19    int *z = &y;
20    *z = *x;
```

Storage

- Variables need a place to live in memory
- Get ‘allocated’ a physical space in memory (with an address)
- Size of memory allocation depends on datatype
- Get ‘deallocated’ to release the space in memory



The stack



Stack stores active functions & local variables

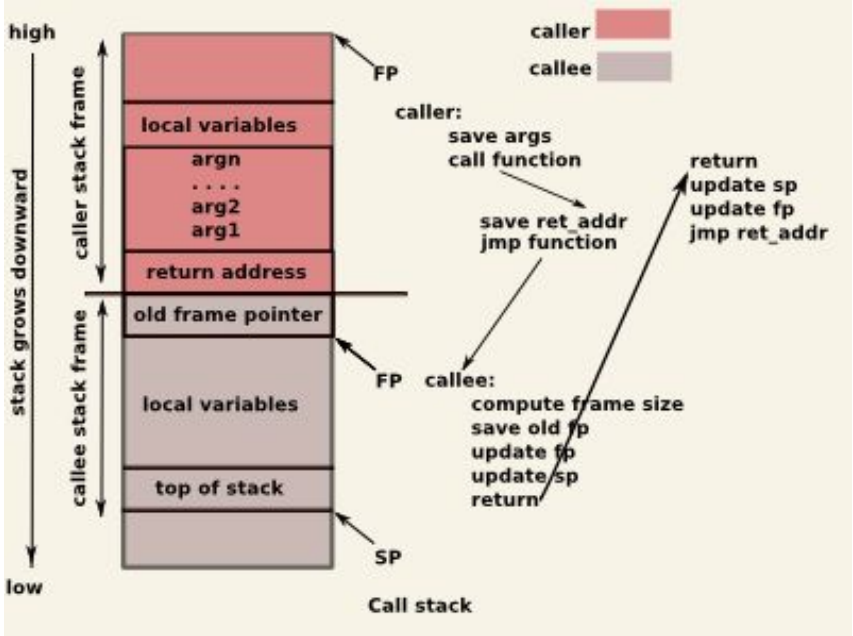
Frames deleted when function returns

Local variables do not persist

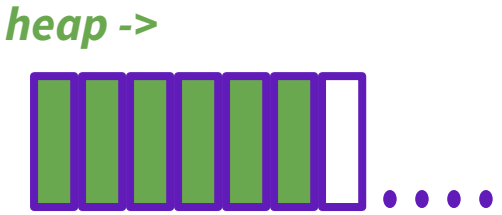
Local variables must have defined size

Can not make run-time adjustments

(Arrays must have length)



The Heap



- Gives us flexible space
- Allocated at run time, with current space requirements
- Persistent until specifically free-ed
- User allocates memory with malloc:

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

Allocate memory block
Allocates a block of size bytes of memory, returning a pointer to the beginning of the block.
Returns NULL in failure -> *should always check for NULL before using pointer*

Malloc - Memory Allocation

- malloc is used in a specific way: (T*)malloc(e * sizeof(T))
  - User doesn't need to know sizeof(T) - use sizeof instead of "16".
- Returns a pointer to memory large enough to hold an array of length e with elements of type T
- malloc returns an untyped pointer (void*); the cast (T*) tells C to treat it as a pointer to a block of type T
- If allocation fails (extremely rare, but can happen), returns NULL.
  - Programs must always check.

Initialization

```
int *var = (int*)malloc(1*sizeof(int));
*var = 255;
```

Malloc does not initialize

Must go set initial values manually

Calloc: `void* calloc (size_t num, size_t size);`

Allocate and zero-initialize array

Allocates a block of memory for an array of num elements, each of them size bytes long, and initializes all its bits to zero.

*Malloc is faster

Halfway through with Memory Management

- ➔ We can now allocate memory of any size and have it “live” forever
  - ◆ For example, we can allocate an array and use it indefinitely, independent of the stack
- ➔ Unfortunately, computers do not have infinite memory so “living forever” could be a problem

Garbage Collection

- ➔ Java solution: Conceptually objects live forever, but the system has a garbage collector that finds unreachable objects and reclaims their space
- ➔ C solution: You explicitly free an object’s space by passing a pointer to it to the library function free
  - ◆ Managing heap memory correctly is hard in complex software and is the disadvantage of C-style heap allocation

Freeing Memory

Deallocate memory block

```
void free (void* ptr);
```

- **A block of memory previously allocated by a call to malloc, calloc or realloc is deallocated, making it available again for further allocations.**
- **If ptr does not point to a block of memory allocated with the above functions, it causes undefined behavior.**
- **If ptr is a null pointer, the function does nothing.**
- **Notice that this function does not change the value of ptr itself, hence it still points to the same (now invalid) location.**

Example

```
int *p = (int*)malloc(sizeof(int));
p = NULL; /* LEAK! - lost address */
int *q = (int*)malloc(sizeof(int));
free(q);
free(q); /* Best case: crash */
int *r = (int*)malloc(sizeof(int));
free(r);
int *s = (int*)malloc(sizeof(int));
*s = 19;
*r = 17; /* Best case: crash */
```

If foo returns a pointer, can the caller free the memory? (Who owns that pointer?)

If bar gets two pointers, can it free one, or both?

Rules

- For every run-time call to malloc there should be one runtime call to free
- If you “lose all pointers” to an object, you can’t ever call free (a leak)!
- Think hard before re-assigning a pointer; where is it pointing?
- If you “use an object after it’s freed” (or free it twice), you used a dangling pointer!
    - Note: It’s possible but rare to use up too much memory without creating “leaks via no more pointers to an object”

Arrays again

"A reference to an object of type array-of-T which appears in an expression decays (with three exceptions) into a pointer to its first element; the type of the resultant pointer is pointer-to-T."

http://c-faq.com/aryptr/aryptrequiv.html

Right: x is the array, which decays to a pointer to an int and &x returns a pointer to the entire array.

```
void f1(int* p) {  // takes a pointer
  *p = 5;
}
int* f2() {
    int x[3];     // x on stack, is pointer
    x[0] = 5;
    (&x)[0] = 5; // address of x, points to
                 // same place but different T
    *x = 5;      // put value at location x
    *(x+0) = 5;  // Also put value at x
    f1(x);
    f1(&x);     // wrong - watch types!
    x = &x[2]; // No!  X isn't really a pointer
    int *p = &x[2];
    return x; // correct type, but is a
              // dangling pointer
}
```

Pointer arithmetic

- If p has type T* or T[] and *p has type T
- If p points to one item of type T, p+1 points to a place in memory for the next item of type T
  - So, p[0] is one item of type T, p+i = p[i]
- T[] always has type T*, even if it is declared as T[]
  - Implicit array promotion

    Result: Arrays are always passed by reference, not by value. (The information passed is the address of where the values are stored.)

Array allocation & manipulation

```
int main() {
  int len = 7;
  int *mainarray;
  mainarray = (int*) malloc (len * sizeof(int));

  for(int i = 0; i < len; i++) {
    mainarray[i] = i;
  }
```

```
26    for(int i = 0; i < len; i++) {
27      printf("Value of myarray[%d] is: %d \n", i, *ind);
28      ind++;
29    }
```

Valgrind

```
$valgrind program arguments
$valgrind --leak-check=full ./dangling
```

Tool used to analyze memory usage (and other things) Catches pointer errors during execution Prints summary of heap usage, including details of memory leaks

Valgrind

```
$valgrind --leak-check=full ./arrdyn
```

The process (a running program) has a single address space for code, globals, the stack, & the heap

When the process exits, the entire address space is freed.

OK to rely on this in many cases

However, good practice to provide mechanism to free any memory allocated in a package, allowing potential clients to release code if desired

https://chortle.ccsu.edu/CPuzzles/CPuzzlesMain.html