UC Berkeley
Teaching Professor
Dan Garcia

CS61C

Great Ideas in Computer Architecture

(a.k.a. Machine Structures)

UC Berkeley
Professor
Bora Nikolić

Introduction to the C Programming Language

Berkeley
UNIVERSITY OF CALIFORNIA

cs61c.org

Garcia, Nikolić

Memory

Dynamic Memory Allocation (1/4)

- C has operator `sizeof()` which gives size in bytes (of type or variable)
- Assume size of objects can be misleading and is bad style, so use `sizeof(type)`
  - Many years ago an `int` was 16 bits, and programs were written with this assumption.
  - What is the size of integers now?
- "`sizeof`" knows the size of arrays:

```
int ar[3]; // Or:   int ar[] = {54, 47, 99}
sizeof(ar) → 12
```

  - …as well for arrays whose size is determined at run-time:

```
int n = 3;
int ar[n]; // Or: int ar[fun_that_returns_3()];
sizeof(ar) → 12
```

Dynamic Memory Allocation (2/4)

- **To allocate room for something new to point to, use `malloc()` (with the help of a typecast and `sizeof`):**

```
ptr = (int *) malloc (sizeof(int));
```

  - Now, `ptr` points to a space somewhere in memory of size (`sizeof(int)`) in bytes.
  - `(int *)` simply tells the compiler what will go into that space (called a typecast).
- **`malloc` is almost never used for 1 var**
- `ptr = (int *) malloc (n*sizeof(int));`
  - This allocates an array of n integers.

Dynamic Memory Allocation (3/4)

- Once `malloc()` is called, the memory location contains garbage, so don't use it until you've set its value.
- After dynamically allocating space, we must dynamically free it:
  - `free(ptr);`
- Use this command to clean up.
  - Even though the program frees all memory on exit (or when main returns), don't be lazy!
  - You never know when your main will get transformed into a subroutine!

Dynamic Memory Allocation (4/4)

- **The following two things will cause your program to crash or behave strangely later on, and cause VERY VERY hard to figure out bugs:**
  - `free()`ing the same piece of memory twice
  - calling `free()` on something you didn't get back from `malloc()`
- **The runtime does not check for these mistakes**
  - Memory allocation is so performance-critical that there just isn't time to do this
  - The usual result is that you corrupt the memory allocator's internal structure
  - You won't find out until much later on, in a totally unrelated part of your code!

Managing the Heap: `realloc(p, size)`

- Resize a previously allocated block at `p` to a new size
- If `p` is NULL, then `realloc` behaves like `malloc`
- If size is 0, then `realloc` behaves like `free`, deallocating the block from the heap
- Returns new address of the memory block; NOTE: it is likely to have moved!

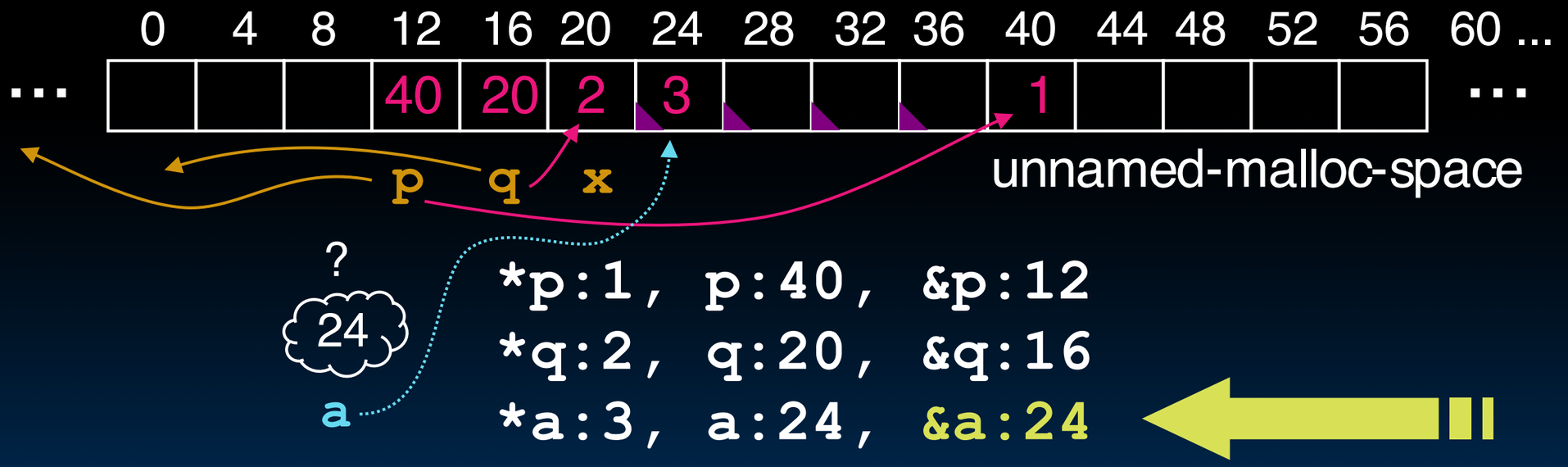
```
int *ip;
ip = (int *) malloc(10*sizeof(int));
/* always check for ip == NULL */
… … …
ip = (int *) realloc(ip,20*sizeof(int));
/* always check NULL, contents of first 10
elements retained */
… … …
realloc(ip,0); /* identical to free(ip) */
```

Arrays not implemented as you'd think

```
void foo() {
    int *p, *q, x;
    int a[4];
    p = (int *)
    malloc (sizeof(int));
    q = &x;
```

```
*p = 1; // p[0] would also work here
printf("*p:%u, p:%u, &p:%u\n", *p, p, &p);
*q = 2; // q[0] would also work here
printf("*q:%u, q:%u, &q:%u\n", *q, q, &q);
*a = 3; // a[0] would also work here
printf("*a:%u, a:%u, &a:%u\n", *a, a, &a);
```

```
}
```



K&R: "An array name is not a variable"

Mini-summary

- **Pointers and arrays are virtually same**
- **C knows how to increment pointers**
- **C is an efficient language, with little protection**
  - Array bounds not checked
  - Variables not automatically initialized
- **Use handles to change pointers**
- **Dynamically allocated heap memory must be manually deallocated in C.**
  - Use `malloc()` and `free()` to allocate and deallocate memory from heap.
- **(Beware) The cost of efficiency is more overhead for the programmer.**
  - "C gives you a lot of extra rope, don't hang yourself with it!"

Linked List Example

Linked List Example

- Let's look at an example of using structures, pointers, `malloc()`, and `free()` to implement a linked list of strings.

```
struct Node {
    char *value;
    struct Node *next;
};
typedef struct Node *List;

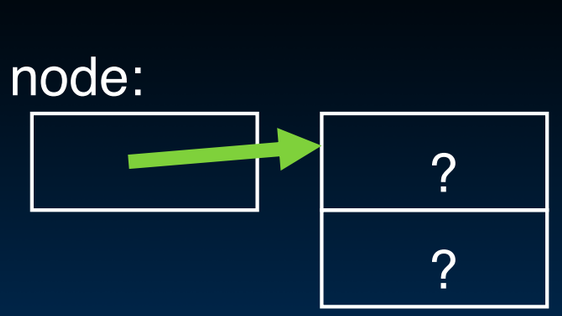
/* Create a new (empty) list */
List ListNew(void)
{ return NULL; }
```

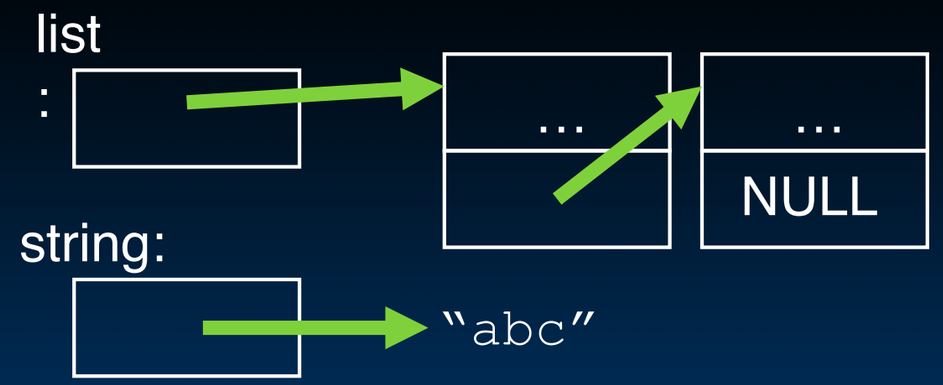
Linked List Example

```
/* add a string to an existing list */
List list_add(List list, char *string)
{
  struct Node *node =
    (struct Node*) malloc(sizeof(struct Node));
  node->value =
    (char*) malloc(strlen(string) + 1);
  strcpy(node->value, string);
  node->next = list;
  return node;
}
```

Linked List Example

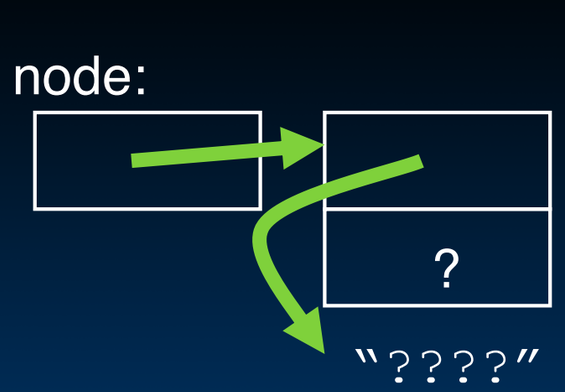
```
/* add a string to an existing list */
List list_add(List list, char *string)
{
  struct Node *node =
    (struct Node*) malloc(sizeof(struct Node));
  node->value =
    (char*) malloc(strlen(string) + 1);
  strcpy(node->value, string);
  node->next = list;
  return node;
}
```

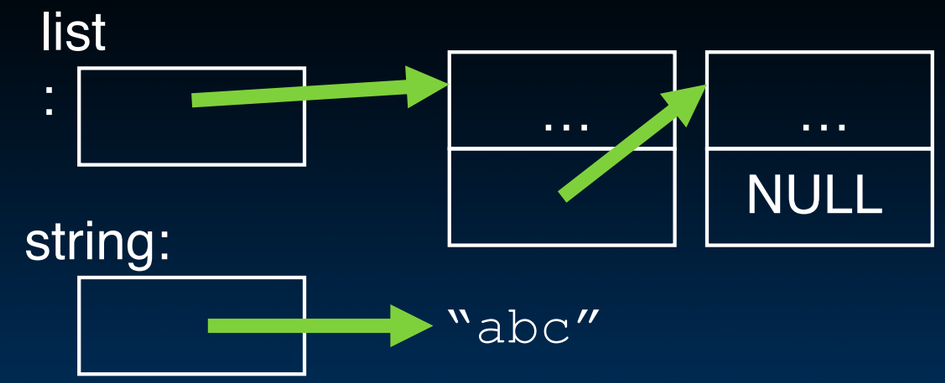




Linked List Example

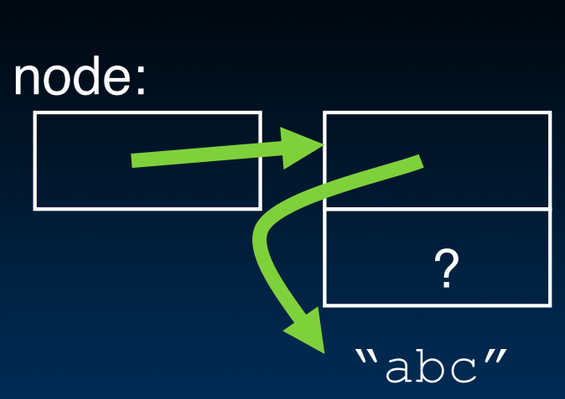
```
/* add a string to an existing list */
List list_add(List list, char *string)
{
  struct Node *node =
    (struct Node*) malloc(sizeof(struct Node));
  node->value =
    (char*) malloc(strlen(string) + 1);
  strcpy(node->value, string);
  node->next = list;
  return node;
}
```

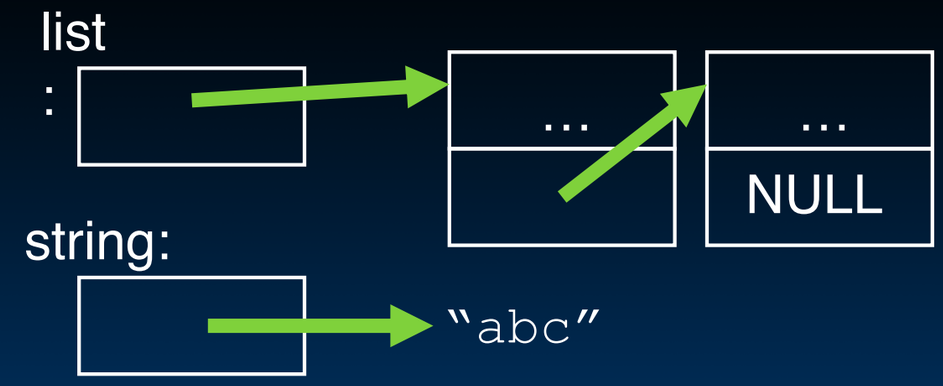




Linked List Example

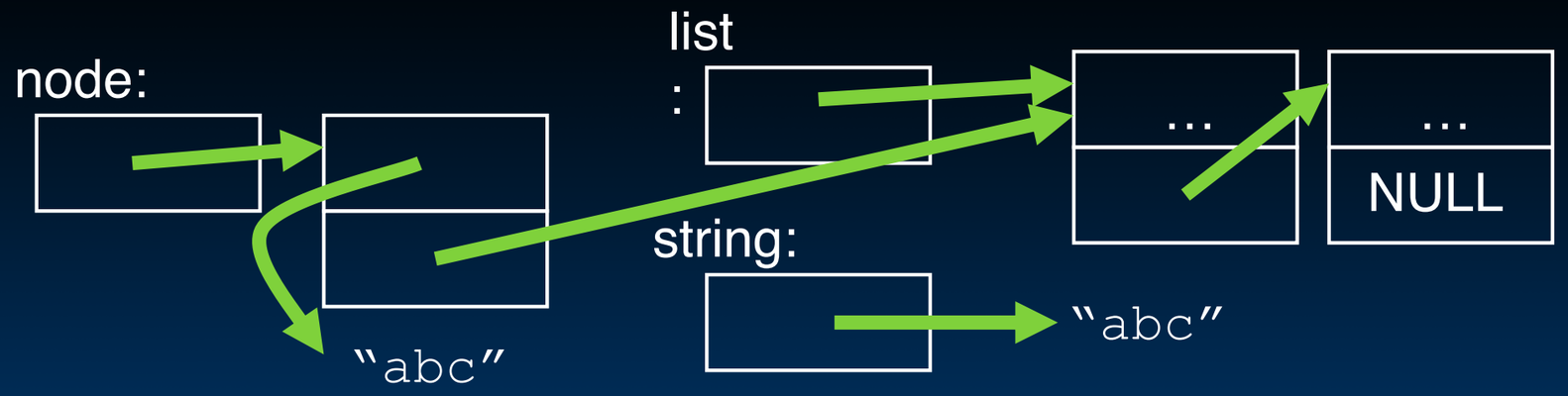
```
/* add a string to an existing list */
List list_add(List list, char *string)
{
  struct Node *node =
    (struct Node*) malloc(sizeof(struct Node));
  node->value =
    (char*) malloc(strlen(string) + 1);
  strcpy(node->value, string);
  node->next = list;
  return node;
}
```





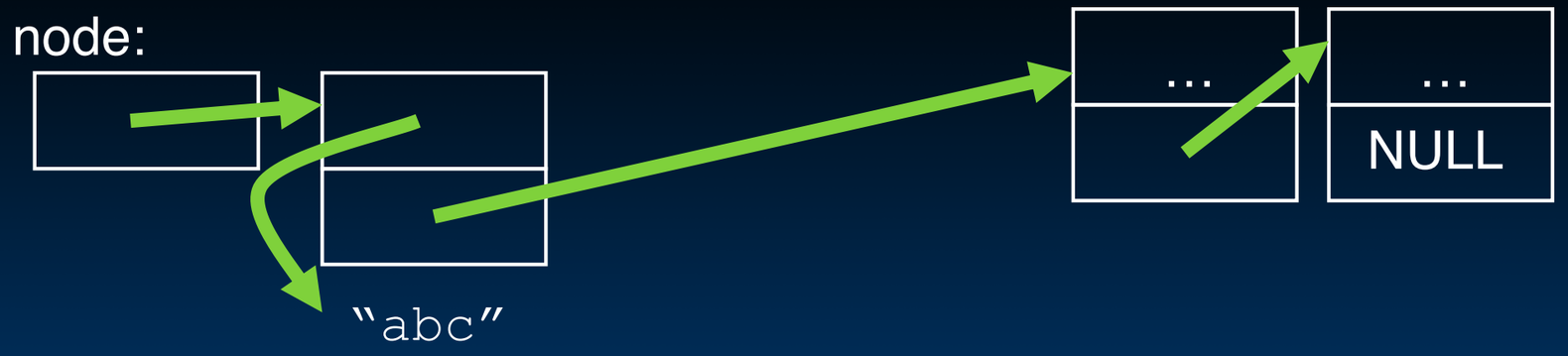
Linked List Example

```
/* add a string to an existing list */
List list_add(List list, char *string)
{
  struct Node *node =
    (struct Node*) malloc(sizeof(struct Node));
  node->value =
    (char*) malloc(strlen(string) + 1);
  strcpy(node->value, string);
  node->next = list;
  return node;
}
```



Linked List Example

```
/* add a string to an existing list */
List list_add(List list, char *string)
{
  struct Node *node =
    (struct Node*) malloc(sizeof(struct Node));
  node->value =
    (char*) malloc(strlen(string) + 1);
  strcpy(node->value, string);
  node->next = list;
  return node;
}
```



Memory Locations

Don't forget the globals!

- **What is stored?**
  - Structure declaration does not allocate memory
  - Variable declaration does allocate memory
- **So far we have talked about several different ways to allocate memory for data:**
  - Declaration of a local variable

```
int i; struct Node list; char *string; int ar[n];
```

  - "Dynamic" allocation at runtime by calling allocation function (alloc).

```
ptr = (struct Node *) malloc (sizeof(struct Node)*n);
```

- **One more possibility exists…**
  - Data declared outside of any procedure (i.e., before `main`).
  - Similar to #1 above, but has "global" scope.

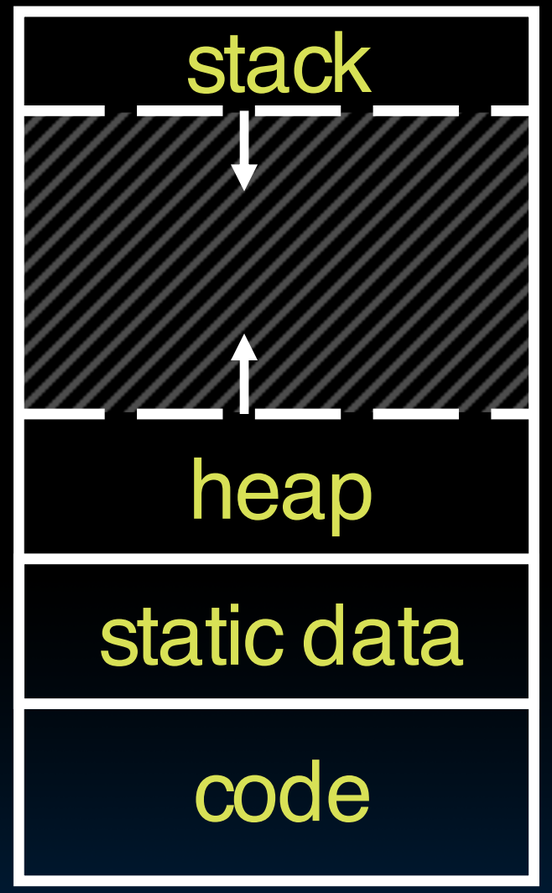
```
int myGlobal;
main() {
}
```

C Memory Management

- **C has 3 pools of memory**
  - Static storage: global variable storage, basically permanent, entire program run
  - The Stack: local variable storage, parameters, return address (location of "activation records" in Java or "stack frame" in C)
  - The Heap (dynamic malloc storage): data lives until deallocated by programmer
- **C requires knowing where objects are in memory, otherwise things don't work as expected**
  - Java hides location of objects

Normal C Memory Management

- **A program's address space contains 4 regions:**
  - stack: local variables, grows downward
  - heap: space requested for pointers via `malloc()` ; resizes dynamically, grows upward
  - static data: variables declared outside main, does not grow or shrink
  - code: loaded when program starts, does not change



For now, OS somehow prevents accesses between stack and heap (gray hash lines). Wait for virtual memory

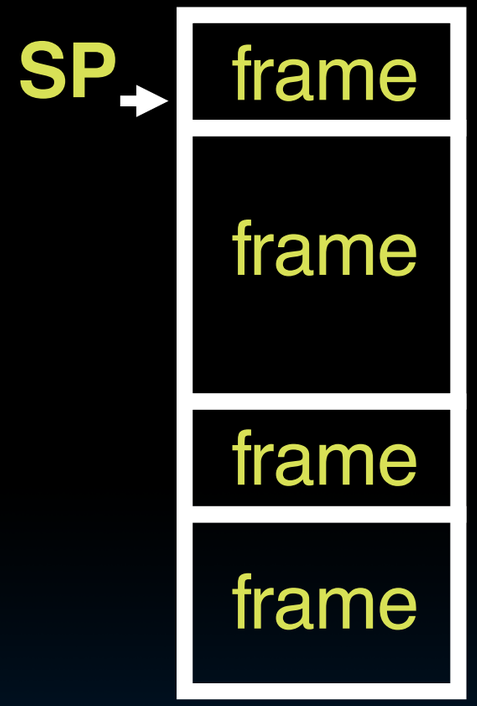
Where are variables allocated?

- If declared **outside** a procedure (global), allocated in "static" storage
- If declared **inside** procedure (local), allocated on the "stack" and **freed when procedure returns**.
  - NB: `main()` is a procedure

```
int myGlobal;
main() {
   int myTemp;
}
```

The Stack

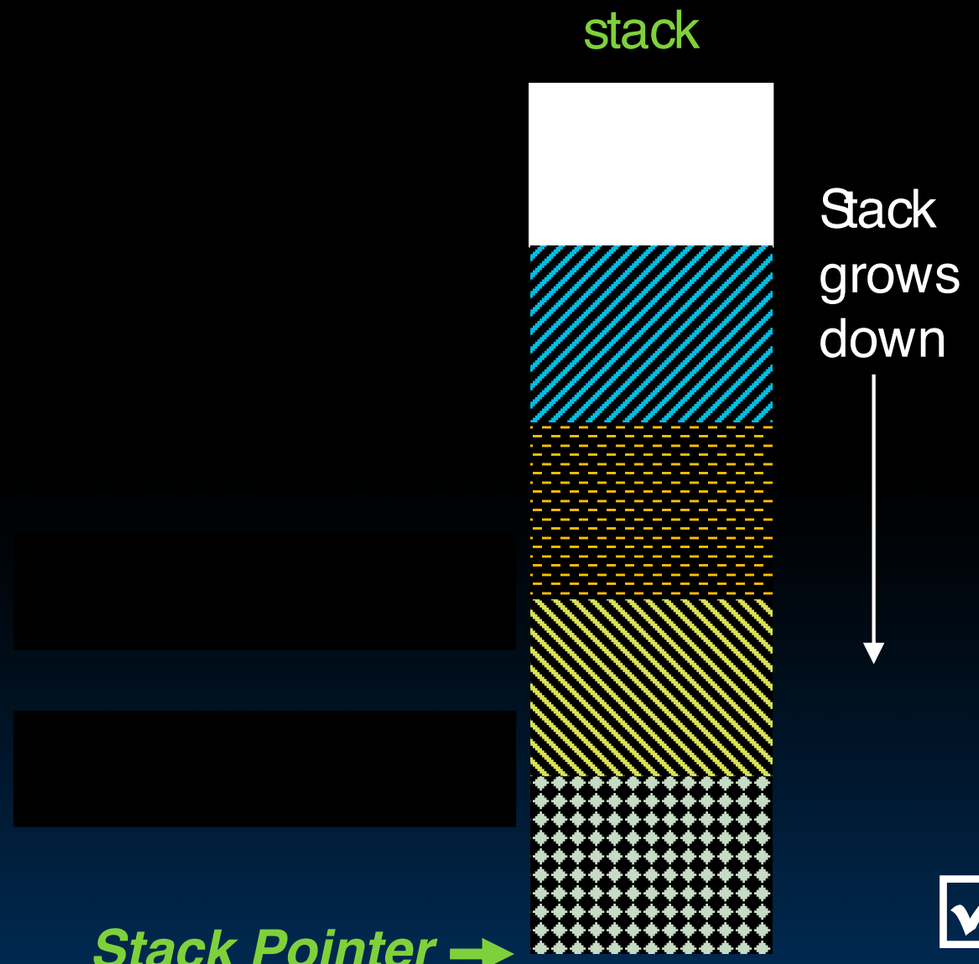
- **Stack frame includes:**
  - Return "instruction" address
  - Parameters
  - Space for other local variables
- **Stack frames contiguous blocks of memory; stack pointer tells where top stack frame is**
- **When procedure ends, stack frame is tossed off the stack; frees memory for future stack frames**



Stack

- Last In, First Out (LIFO) data structure

```
main ()
{ a(0);
}
  void a (int m)
  { b(1);
  }
    void b (int n)
    { c(2);
    }
      void c (int o)
      { d(3);
      }
        void d (int p)
        {
        }
```



Memory Management

The Heap (Dynamic memory)

- **Large pool of memory, not allocated in contiguous order**
  - back-to-back requests for heap memory could result blocks very far apart
  - where Java new command allocates memory
- **In C, specify number of bytes of memory explicitly to allocate item**

```
int *ptr;
ptr = (int *) malloc(sizeof(int));
/* malloc returns type (void *),
so need to cast to right type */
```

  - `malloc()`: Allocates raw, uninitialized memory from heap

Memory Management

- **How do we manage memory?**
- **Code, Static storage are easy:**
  - they never grow or shrink
- **Stack space is also easy:**
  - stack frames are created and destroyed in last-in, first-out (LIFO) order
- **Managing the heap is tricky:**
  - memory can be allocated / deallocated at any time

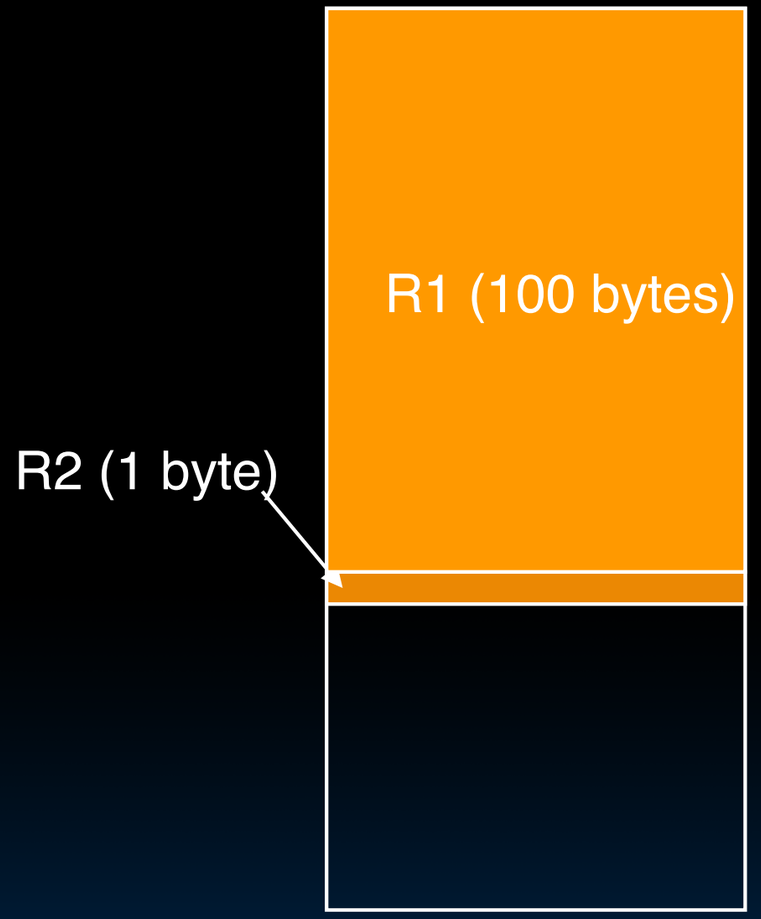
Heap Management Requirements

- Want `malloc()` and `free()` to run quickly
- Want minimal memory overhead
- Want to avoid fragmentation* – when most of our free memory is in many small chunks
  - In this case, we might have many free bytes but not be able to satisfy a large request since the free bytes are not contiguous in memory.

* This is technically called external fragmention

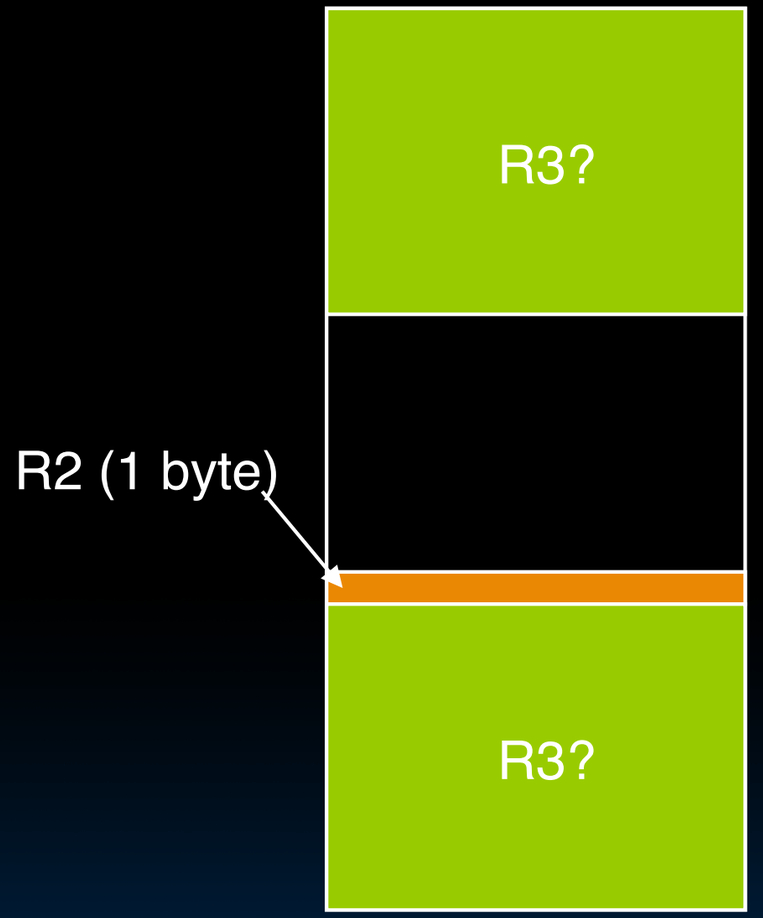
Heap Management

- **An example**
  - Request R1 for 100 bytes
  - Request R2 for 1 byte
  - Memory from R1 is freed
  - Request R3 for 50 bytes



Heap Management

- **An example**
  - Request R1 for 100 bytes
  - Request R2 for 1 byte
  - Memory from R1 is freed
  - Request R3 for 50 bytes



K&R Malloc/Free Implementation

- From Section 8.7 of K&R
  - Code in the book uses some C language features we haven't discussed and is written in a very terse style, don't worry if you can't decipher the code
- Each block of memory is preceded by a header that has two fields:
  size of the block and
  a pointer to the next block
- All free blocks are kept in a circular linked list, the pointer field is unused in an allocated block

K&R Implementation

- `malloc()` searches the free list for a block that is big enough. If none is found, more memory is requested from the operating system. If what it gets can't satisfy the request, it fails.
- `free()` checks if the blocks adjacent to the freed block are also free
  - If so, adjacent free blocks are merged (coalesced) into a single, larger free block
  - Otherwise, freed block is just added to the free list

Choosing a block in `malloc()`

- **If there are multiple free blocks of memory that are big enough for some request, how do we choose which one to use?**
  - best-fit: choose the smallest block that is big enough for the request
  - first-fit: choose the first block we see that is big enough
  - next-fit: like first-fit but remember where we finished searching and resume searching from there

And in conclusion…

- **C has 3 pools of memory**
  - Static storage: global variable storage, basically permanent, entire program run
  - The Stack: local variable storage, parameters, return address
  - The Heap (dynamic storage): malloc() grabs space from here, free() returns it.
- **`malloc()` handles free space with freelist**
- **Three ways to find free space when given a request:**
  - First fit (find first one that's free)
  - Next fit (same as first, but remembers where left off)
  - Best fit (finds most "snug" free space)

When Memory Goes Bad

Pointers in C

- **Why use pointers?**
  - If we want to pass a huge struct or array, it's easier / faster / etc to pass a pointer than the whole thing.
  - In general, pointers allow cleaner, more compact code.
- **So what are the drawbacks?**
  - Pointers are probably the single largest source of bugs in software, so be careful anytime you deal with them.
  - Dangling reference (use ptr before malloc)
  - Memory leaks (tardy free, lose the ptr)

Writing off the end of arrays...

```
int *foo = (int *) malloc(sizeof(int) * 100);
int i;
....
for(i = 0; i <= 100; ++i) {
   foo[i] = 0;
}
```

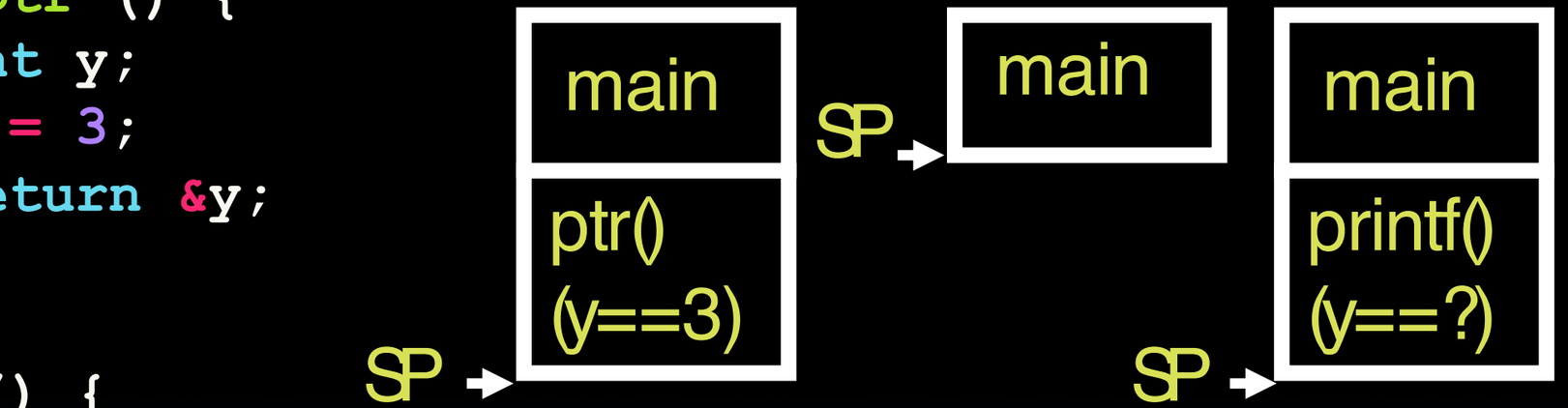
- **Corrupts other parts of the program...**
  - Including internal C data
- **May cause crashes later**

Returning Pointers into the Stack

- Pointers in C allow access to deallocated memory, leading to hard-to-find bugs !

```
int *ptr () {
    int y;
    y = 3;
    return &y;
};

main () {
    int *stackAddr, content;
    stackAddr = ptr();
    content = *stackAddr;
    printf("%d", content); /* 3 */
    content = *stackAddr;
    printf("%d", content); /*13451514 */
};
```



Use After Free

- **When you keep using a pointer..**

```
struct foo *f
....
f = malloc(sizeof(struct foo));
....
free(f)
....
bar(f->a);
```

- **Reads after the free may be corrupted**
  - As something else takes over that memory. Your program will probably get wrong info!
- **Writes corrupt other data!**
  - Uh oh... Your program crashes later!

Forgetting `realloc` Can Move Data...

- When you `realloc` it can copy data...
  - ```
    struct foo *f = malloc(sizeof(struct foo) * 10);
    ...
    struct foo *g = f;
    ....
    f = realloc(sizeof(struct foo) * 20);
    ```
- Result is `g` *may* now point to invalid memory
  - So reads may be corrupted and writes may corrupt other pieces of memory
    ```

Freeing the Wrong Stuff...

- If you `free()` something never `malloc`'ed()
  - Including things like

```
struct foo *f = malloc(sizeof(struct foo) * 10)
...
f++;
...
free(f)
```

- `malloc` or `free` may get confused..
  - Corrupt its internal storage or erase other data...

Double-Free...

- Eg.,

```
struct foo *f = (struct foo *)
    malloc(sizeof(struct foo) * 10);
...
free(f);
...
free(f);
```

- May cause either a use after `free` (because something else called `malloc()` and got that data) or corrupt `malloc`'s data (because you are no longer freeing a pointer called by `malloc`)

Losing the initial pointer! (Memory Leak)

- What is wrong with this code?

```
int *plk = NULL;
void genPLK() {
  plk = malloc(2 * sizeof(int));
  … …
    plk++;
}
```

This MAY be a memory leak if we don't keep somewhere else a copy of the original malloc'ed pointer

Valgrind to the rescue…

- **Valgrind slows down your program by an order of magnitude, but…**
  - It adds a tons of checks designed to catch most (but not all) memory errors
    - Memory leaks
    - Misuse of free
    - Writing over the end of arrays
- **Tools like Valgrind are absolutely essential for debugging C code**

And In Conclusion, …

- **C has three main memory segments in which to allocate data:**
  - Static Data: Variables outside functions
  - Stack: Variables local to function
  - Heap: Objects explicitly malloc-ed/free-d.
- **Heap data is biggest source of bugs in C code**