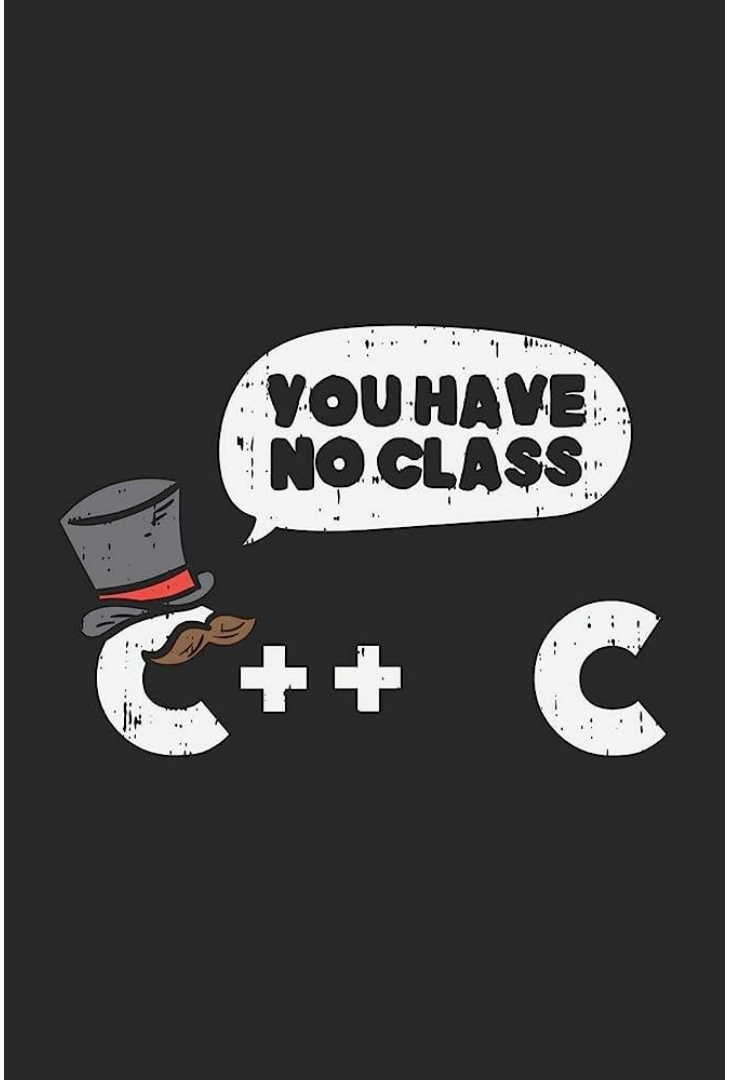
CS61C: Great Ideas in Computer Architecture (Machine Structure)

Instructors: Rosalie Fang, Charles Hong, Jero Wang

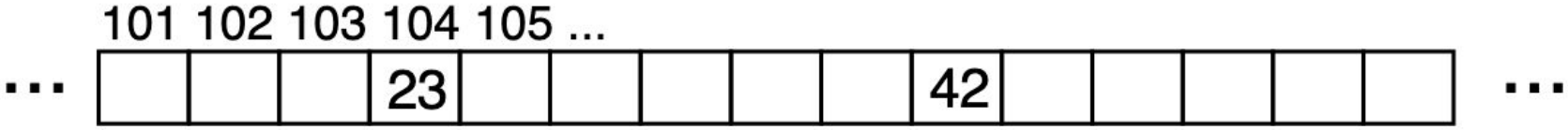
Last Time



Introduction to Pointers

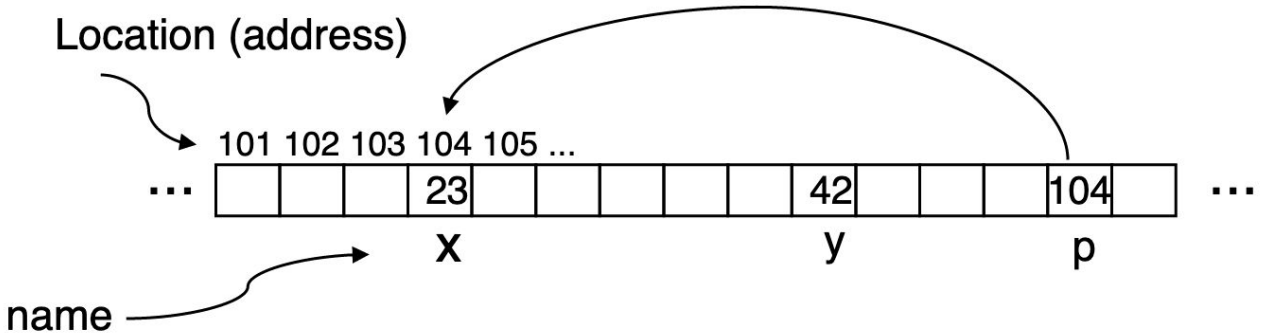
Address vs. Value

- Consider memory to be a single huge array:
  - Each cell of the array has an address associated with it.
  - Each cell also stores some value.
  - Do you think they use signed or unsigned numbers? Negative address?!
- Don’t confuse the *address* referring to a memory location with the *value* stored in that location.
  - For now, the abstraction lets us think we have access to ∞ memory, numbered from 0…



Pointers

- An address refers to a particular memory location. In other words, it points to a memory location.
- *Pointer*: A variable that contains the address of a variable.



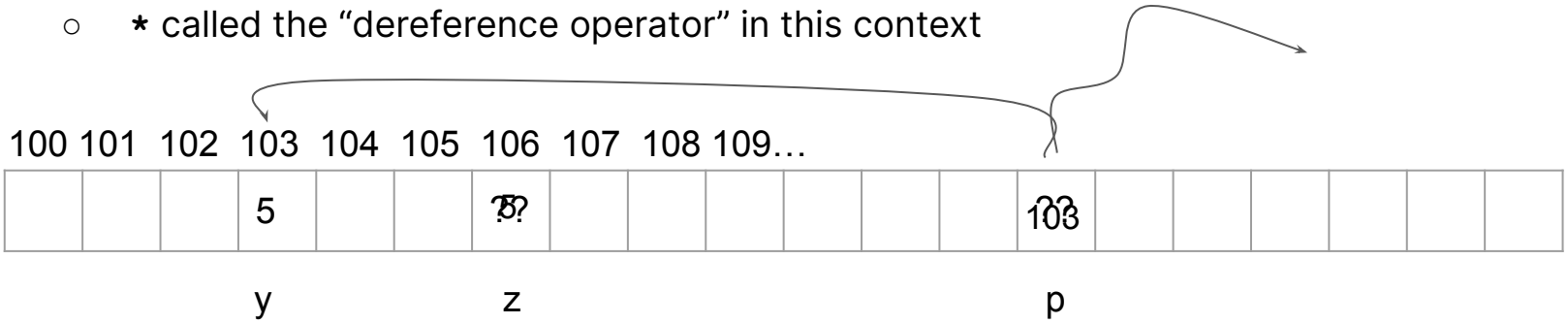
Syntax

C Syntax: Variable Declarations

- Similar to Java, but with a few minor but important differences
  - All variable declarations must appear before they are used
  - All must be at the beginning of a block.
  - A variable may be initialized in its declaration;
    if not, it holds garbage! **`E.g. int x;`**
  - the contents are undefined…
- Examples of declarations:
  - Correct: **`{ int a = 0, b = 10;`** …
  - Incorrect in ANSI C: **`for (int i=0;`** …
  - Correct in C99 (and beyond): **`for (int i=0;`**…

Pointer Syntax

- **`int *p;`**
  - Tells compiler that variable **`p`** is address of an **`int`**
- **`p = &y;`**
  - Tells compiler to assign address of **`y`** to **`p`**
  - **`&`** called the “address operator” in this context
- **`z = *p;`**
  - Tells compiler to assign value at address in **`p`** to **`z`**
  - **`*`** called the “dereference operator” in this context



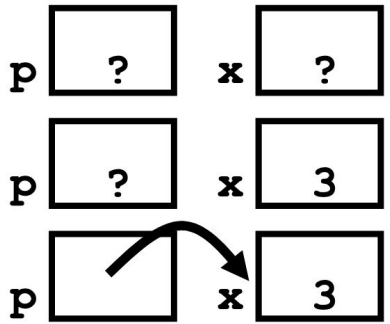
Pointers

- How to create a pointer:
  `&` operator: get address of a variable

```
int *p, x;

x = 3;

p = &x;
```

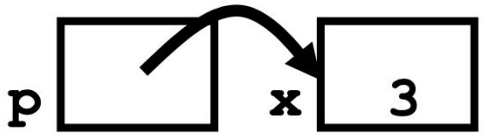


- How get a value pointed to?
  `*` “dereference operator”: get value pointed to
- `printf("p points to %d\n", *p);`

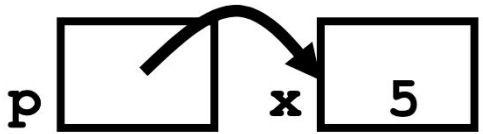
Note the “*” gets used 2 different ways in this example. In the declaration to indicate that p is going to be a pointer, and in the printf to get the value pointed to by p.

Pointers

- How to change a variable pointed to?
- Use dereference `*` operator on left of `=`




```
*p = 5;
```



Common Mistakes

An Important Note: Undefined Behavior...

- A lot of C has “Undefined Behavior”
  - This means it is often unpredictable behavior
  - It will run one way on one computer...
  - But some other way on another
  - Or even just be different each time the program is executed!
- Often characterized as “Heisenbugs”
  - Bugs that seem random/hard to reproduce, and seem to disappear or change when debugging
  - Cf. “Bohrbugs” which are repeatable

More C Pointer Dangers

- Declaring a pointer just allocates space to hold the pointer – it does not allocate something to be pointed to!
- Local variables in C are not initialized, they may contain anything.
- What does the following code do?
- ```
  void f()
  {
      int *ptr;
      *ptr = 5;
  }
  ```
  ```

Pointers in C … The Good, Bad, and the Ugly

- Why use pointers?
  - If we want to pass a large struct or array, it’s easier / faster / etc. to pass a pointer than the whole thing
  - Otherwise we’d need to copy a huge amount of data
  - In general, pointers allow cleaner, more compact code
- So what are the drawbacks?
  - Pointers are probably the single largest source of bugs in C, so be careful anytime you deal with them
  - Most problematic with dynamic memory management—coming up next time
  - Dangling references and memory leaks

Effective Usage

Pointers and Parameter Passing

- Java and C pass parameters “by value”
  - procedure/function/method gets a copy of the parameter, so changing the copy cannot change the original
- ```
  void addOne (int x) {
      x = x + 1;
  }
  int y = 3;
  addOne(y);
  ```
- `y` is still = 3

- How to get a function to change a value?
- ```
  void addOne (int *p) {
      *p = *p + 1;
  }
  int y = 3;
  addOne(&y);
  ```
- `y` is now = 4

Pointers

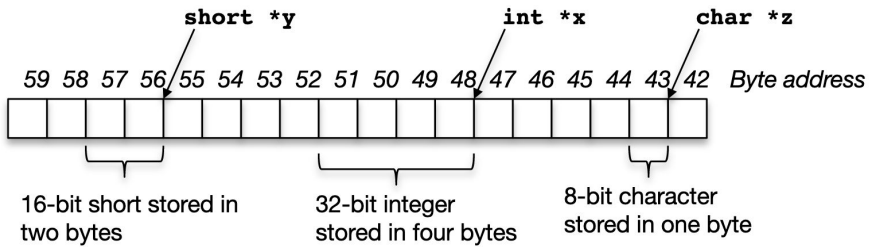
- Pointers are used to point to any data type (**`int`**, **`char`**, a **`struct`**, etc.).
- Normally a pointer can only point to one type (**`int`**, **`char`**, a **`struct`**, etc.).
  - **`void *`** is a type that can point to anything (generic pointer)
  - Use sparingly to help avoid program bugs… and security issues… and a lot of other bad things!
- You can even have pointers to functions…
  - **`int (*fn) (void *, void *) = &foo`**
  - **`fn`** is a function that accepts two **`void *`** pointers and returns an int and is initially pointing to the function foo.
  - **`(*fn)(x, y)`** will then call the function

NULL pointers

- The pointer of all 0s is special
  - The "NULL" pointer, like in Java, python, etc...
  - `int *p = NULL;`
  - `int *p = 0;  /* don't use this one manually! */`
- If you write to or read a null pointer, your program should crash
- Since "0 is false", its very easy to do tests for null:
  - `if(!p) { /* P is a null pointer */ }`
  - `if(q) { /* Q is not a null pointer */ }`

Pointing to Different Size Objects

- Modern machines are “byte-addressable”
  - Hardware’s memory composed of 8-bit storage cells, each has a unique address
- A C pointer is just abstracted memory address
- Type declaration tells compiler how many bytes to fetch on each access through pointer
  - E.g., 32-bit integer stored in 4 consecutive 8-bit bytes
- But we actually want “word alignment”
  - Some processors will not allow you to address 32b values without being on 4 byte boundaries
  - Others will just be very slow if you try to access “unaligned” memory.



Pointer Arithmetic

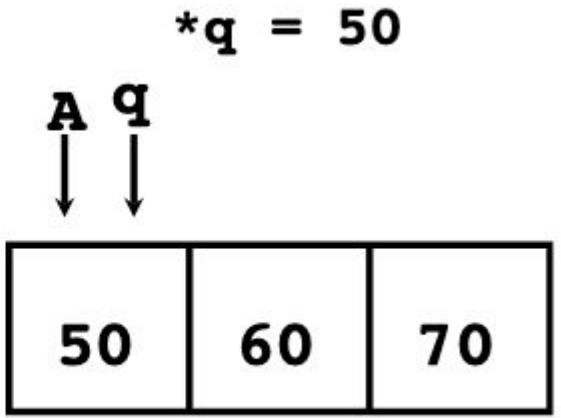
- **`pointer + n`**
  - Adds **`n*sizeof`**(“whatever pointer is pointing to”) to the memory address
- **`pointer - n`**
  - Subtract **`n*sizeof`**(“whatever pointer is pointing to”) to the memory address

Changing Pointers

- What if we need to change the pointer itself?
  - Changing the value of a variable?
  - Changing the value of the pointer pointing to the variable?

```
void inc_ptr(int *p)
{    p =  p + 1;   }

int A[3] = {50, 60, 70};
int* q = A;
inc_ptr( q);
printf("*q = %d\n", *q);
```

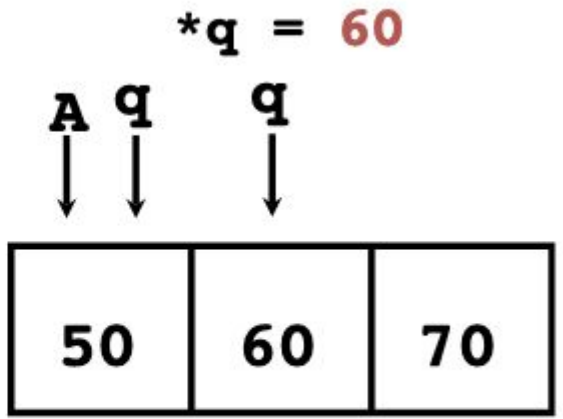


Changing Pointers

- We use a... pointer to a pointer!

```
void inc_ptr(int **h)
{    *h = *h + 1;    }

int A[3] = {50, 60, 70};
int* q = A;
inc_ptr(&q);
printf("*q = %d\n", *q);
```



Pointer Depth is Limitless

- You can have something like this:
  `int **********x;`
- x is a pointer to a pointer to a pointer to a pointer to a pointer to a pointer to a pointer to a pointer to a pointer to a pointer to an integer!

Characteristics of Arrays

Arrays (1/5)

- Declaration:
  - `int ar[2];`
  - ...declares a 2-element integer array
  - An array is really just a block of memory
- Declaration and initialization
  - `int ar[] = {795, 635};`
  - declares and fills a 2-elt integer array
- Accessing elements:
  - `ar[num]`
  - returns the numth element.

Pointers vs Arrays

Arrays (2/5)

- Arrays are (almost) identical to pointers
  - `char *string` and `char string[]` are nearly identical declarations
  - They differ in very subtle ways: incrementing, declaration of filled arrays
- Key Concept: An array variable is a “pointer” to the first element.

Arrays (3/5)

- Consequences:
  - `ar` is an array variable but looks like a pointer in many respects (though not all)
  - `ar[0]` is the same as `*ar`
  - `ar[2]` is the same as `*(ar+2)`
  - We can use pointer arithmetic to access arrays more conveniently.
- Declared arrays are only allocated while the scope is valid
  - ```
    char *foo() {
       char string[32]; ...;
       return string;
    }
    ```
  - is incorrect
    ```

Arrays (4/5)

- Array size n; want to access from 0 to n-1, so you should use counter AND utilize a variable for declaration & incr
  - Wrong

```
int i, ar[10];
for(i = 0; i < 10; i++){ ...
```

  - Right

```
int ARRAY_SIZE = 10;
int i, a[ARRAY_SIZE];
for(i = 0; i < ARRAY_SIZE; i++){ ... }
```

- Why? **Single source of truth**
- You're utilizing indirection and avoiding maintaining two copies of the number 10

Arrays (5/5)

- Pitfall: An array in C does *not* know its own length, & bounds not checked!
  - Consequence: We can accidentally access off the end of an array.
  - Solution: We must pass the array *and its size* to a procedure which is going to traverse it.
- Segmentation faults and bus errors:
  - These are *very* difficult to find; be careful!
  - You'll learn how to debug these in lab...

Memory Allocation

Dynamic Allocation

- Memory is allocated on the heap
- All dynamically allocated memory must be manually freed (`free(mem_addr)`)before returning from `main` otherwise memory leaks will occur
  - Memory leaks: occurs when a system incorrectly manages memory, such that memory is still dynamically allocated, but we cannot access it anymore
  - Can occur for a variety of reasons! (e.g. reassigning a variable, returning without free)
- 3 types
  - `malloc`: **m**emory **alloc**ation
  - `calloc`: **c**leared **alloc**ation
  - `realloc`: **re-alloc**ation
- Memory is not always allocated linearly!

Dynamic Allocation Failure

- Think of memory as a band that the system samples chunks from
- If there is a **contiguous** chunk of memory that's at least as large as what's requested, the system will return the pointer (typed memory address) and metadata for the chunk, otherwise a NULL pointer.
- NOTE: even if there is overall enough memory, if the memory addresses are not continuous, it is not considered.
    - What does this mean and why does this occur?

Let's allocate some memory! We'll be colour-coding arrows instead of making a bunch of variables.

Memory request list
Pink: a-sized memory chunk

Request Result
Bumps "start" of free list to the end of pink chunk!

Memory request list
Pink: a-sized memory chunk
Orange: b-sized memory chunk

Request Result
Bumps "start" of free list to the end of orange chunk!

Memory request list
Pink: a-sized memory chunk
Orange: b-sized memory chunk
Yellow: c-sized memory chunk

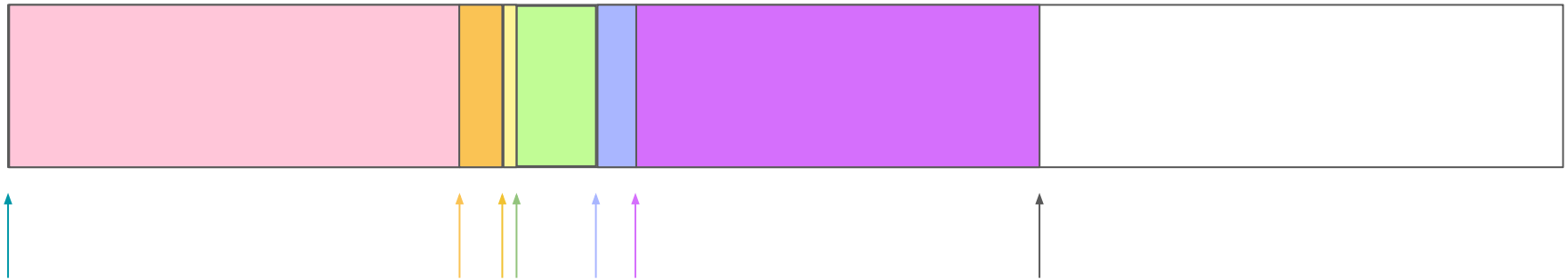
Request Result
Bumps "start" of free list to the end of yellow chunk!

Memory request list
Pink: a-sized memory chunk
Orange: b-sized memory chunk
Yellow: c-sized memory chunk
Green: d-sized memory chunk

Request Result
Bumps "start" of free list to the end of green chunk!

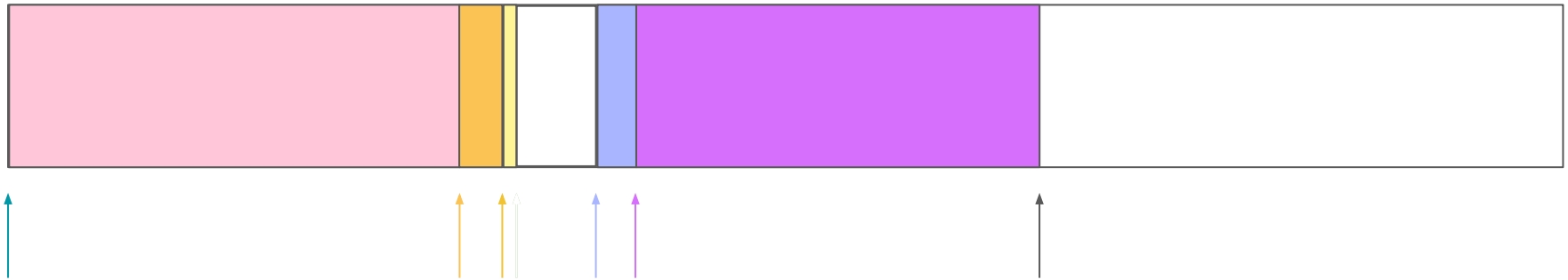
Memory request list
Pink: a-sized memory chunk
Orange: b-sized memory chunk
Yellow: c-sized memory chunk
Green: d-sized memory chunk
Violet: e-sized memory chunk

Request Result
Bumps "start" of free list to the end of violet chunk!



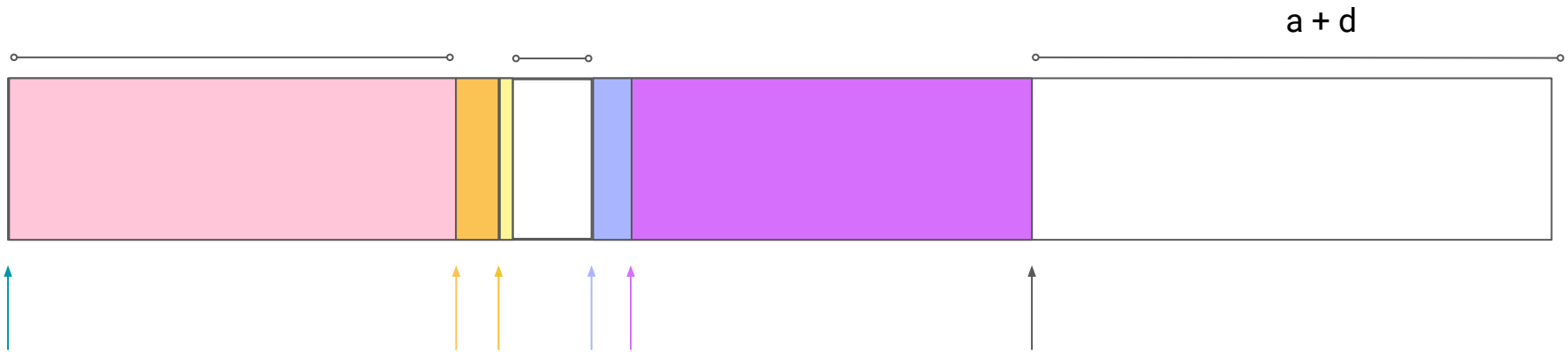
Memory request list
Pink: a-sized memory chunk
Orange: b-sized memory chunk
Yellow: c-sized memory chunk
Green: d-sized memory chunk
Violet: e-sized memory chunk
Purple: f-sized memory chunk

Request Result
Bumps "start" of free list to the end of purple chunk!



Memory request list
Pink: a-sized memory chunk
Orange: b-sized memory chunk
Yellow: c-sized memory chunk
Green: d-sized memory chunk
Violet: e-sized memory chunk
Purple: f-sized memory chunk
free(green)

Request Result
Green chunk becomes white ⇒ free list



Memory request list
Pink: a-sized memory chunk
Orange: b-sized memory chunk
Yellow: c-sized memory chunk
Green: d-sized memory chunk
Violet: e-sized memory chunk
Purple: f-sized memory chunk
free(green)
request: a + d sized chunk

Request Result
Fails! (Why?)

Fragmentation

- What're observations we have about the previous example?
  - Even though we had enough memory, we still weren't able to get it
  - We had a lot of very small memory requests
  - We don't automatically merge all non-used heap memory into one larger chunk
- What would've happened if we'd kept going in this way? (small requests, frequent non-linear frees)
  - Memory would've continued to have tiny slivers
  - We might end up with more holes (Swiss cheese heap?)
- This is considered **fragmentation**!
  - Defined as when most of memory has been allocated in large or smaller (but more frequently, smaller) non-sequential chunks
- Can result in
  - Out-of-memory exceptions, NULL pointer returns

If not linear, then how?

- Method used: finding what's the first available chunk of requested size
  - Pros:
    - Easy to understand: can diagram it easily
    - Easy to implement: just find the last used address and start from there
  - Cons:
    - Not always efficient: allocation can easily result in fragmentation, memory waste
  - Called "first fit"
- Next Fit: allocates memory block according to next free partition fitting request
  - Pros:
    - Fast(er): next addresses to allocate are always directly referrable (no need to search entire free list); O(1) best case, O(n) worst case
  - Cons:
    - Inefficient memory allocation: similar problem to first fit, since next block might not be the optimal one and could cause slivering internal fragmentation

If not linear, then how? (continued)

- Best Fit: allocates memory block according to closest-fitting free partition
  - Pros:
    - Memory efficient: wastes least memory since free block usage is closest in size
  - Cons:
    - Slow allocation: O(n) in search free list
- Worst Fit: allocates memory block in largest free partition
  - Pros:
    - Purposefully causes internal fragmentation, but large enough such that smaller processes can fit inside fragmentation
  - Cons:
    - Slow allocation: O(n) like best fit

Dynamic Allocation in C

Dynamic Memory Allocation (1/4)

- C has operator **`sizeof()`** which gives size in bytes (of type or variable)
- 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`**
- What's the size of a pointer? **`sizeof(int*)`**
  - It depends on the system!!! (Usually 4 or 8)

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!

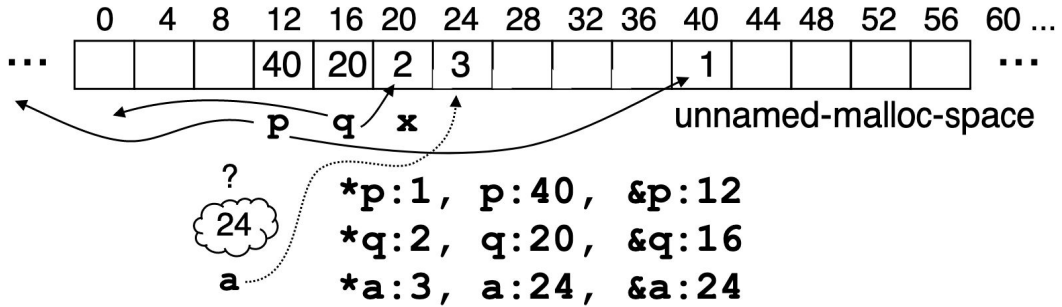
Managing the Heap: `realloc(p, size)`

- 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"

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...

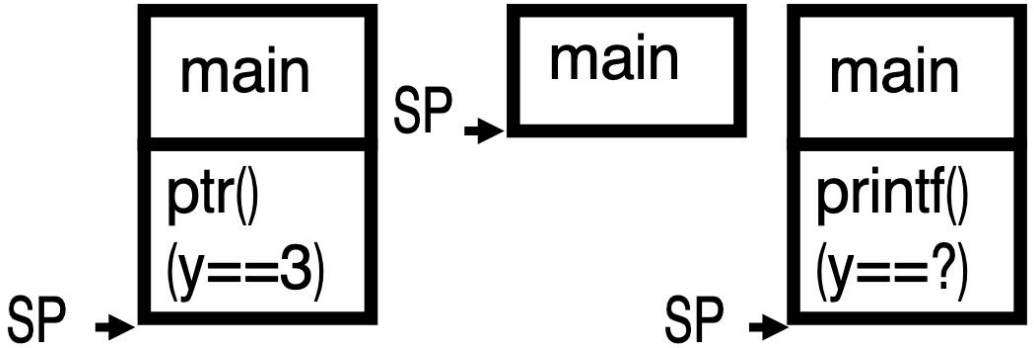
- ```
  int *foo = (int *) malloc(sizeof(int) * 100);
  int i;
  ....
  for(i = 0; i <= 100; ++i) {
     foo[i] = 0;
  }
  ```
- Corrupts other parts of the program...
  - Including internal C data
- May cause crashes later
  ```

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...

- E.g.,

```
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`**)

```
double free or corruption (out)
```

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

How does `int x = -83;` look like in memory?

Endianness

- So far, we've discussed how we store values in binary
  - Ex. We write **`int[] i = {0x6472 6167, 0x7320 6E65, 0x7400 646F}.`**
  - If we assume **`&i == 0xF000 0000`**, then our memory would look something like this:

Address (Last hex digit)	0	1	2	3	4	5	6	7	8	9	A	B
Value	0x64726167				0x73206E65				0x7400646F			

- If we do **`i[1]`**, the compiler adds **`0xF000 0000 + 1 * sizeof(int) = 0xF000 0004`**, then takes the four bytes starting from that address as an integer. This corresponds to 0×73206E65, so it works.
- What happens if we do **`((char*) i)[2]`**?
- Note: These slides follow a convention of using a new 0x prefix for every array element. Thus, **`0x64726167 0x73206E65 0x7400646F`** is an array of 32-bit values, while **`0x64 0x72 0x61 0x67`** is an array of 8-bit values.

Endianness

- So far, we've discussed how we store values in binary
  - Ex. We write `int[] i = {0x6472 6167, 0x7320 6E65, 0x7400 646F}`.
  - If we assume `&i == 0xF000 0000`, then our memory would look something like this:

Address (Last hex digit)	0	1	2	3	4	5	6	7	8	9	A	B
Value	0x64726167				0x73206E65				0x7400646F			
Value	?	?	?	?	?	?	?	?	?	?	?	?

- If we do `((char*) i)[2]`, the compiler adds 0xF000 0000 + 2 * sizeof(char) = 0xF000 0002, then takes the one byte starting from that address as a char. This yields something (since there is data there), but what it yields depends on how the subbytes of our 4-byte int get stored in memory. The way things get stored is known as the endianness of the system.

Big-Endian

- In a big-endian system, we write the Most Significant Byte “first” (that is, in the lower address.

Address (Last hex digit)	0	1	2	3	4	5	6	7	8	9	A	B
Value	0x64726167				0x73206E65				0x7400646F			
Value	0x64	0x72	0x61	0x67	0x73	0x20	0x6E	0x65	0x74	0x00	0x64	0x6F

- If we do **`((char*) i)[2]`**, the compiler adds **`0xF000 0000 + 2 * sizeof(char) = 0xF000 0002`**, then takes the one byte starting from that address as a char. This yields 0×61.
- If we do **`printf((char*) i);`** we would interpret this block of memory as a character array, so we’d get 0×64 0×72 0×61 ..., which when converted to ASCII yields “drags net”

Little-Endian

- In a little-endian system, we write the Least Significant Byte “first” (that is, in the lower address.

Address (Last hex digit)	0	1	2	3	4	5	6	7	8	9	A	B
Value	0x64726167				0x73206E65				0x7400646F			
Value	0x67	0x61	0x72	0x64	0x65	0x6E	0x20	0x73	0x6F	0x64	0x00	0x74

- If we do **`((char*) i)[2]`**, the compiler adds **`0xF000 0000 + 2 * sizeof(char) = 0xF000 0002`**, then takes the one byte starting from that address as a char. This yields 0×72.
- If we do **`printf((char*) i);`** we would interpret this block of memory as a character array, so we’d get 0×67 0×61 0×72 ..., which when converted to ASCII yields “garden sod”

Endianness

- Note: In either case, if we read **`i[1]`**, we end up with our original integer. This is a good checksum to confirm your understanding; if you write a number, you should be able to read that number to get the same thing.

Address (Last hex digit)	0	1	2	3	4	5	6	7	8	9	A	B
Value	0x64726167				0x73206E65				0x7400646F			
Value (Big Endian)	0x64	0x72	0x61	0x67	0x73	0x20	0x6E	0x65	0x74	0x00	0x64	0x6F
Value (Little Endian)	0x67	0x61	0x72	0x64	0x65	0x6E	0x20	0x73	0x6F	0x64	0x00	0x74

- In Big Endian: the compiler adds **`0xF000 0000 + 1 * sizeof(int) = 0xF000 0004`**, then takes the four bytes starting from that address: 0×73 0×20 0×6E 0×65. Since this is big-endian, we combine them with the first address being the MSB, so we get 0×73206E65, which is the correct result

Endianness

- Note: In either case, if we read **`i[1]`**, we end up with our original integer. This is a good checksum to confirm your understanding; if you write a number, you should be able to read that number to get the same thing.

Address (Last hex digit)	0	1	2	3	4	5	6	7	8	9	A	B
Value	0x64726167				0x73206E65				0x7400646F			
Value (Big Endian)	0x64	0x72	0x61	0x67	0x73	0x20	0x6E	0x65	0x74	0x00	0x64	0x6F
Value (Little Endian)	0x67	0x61	0x72	0x64	0x65	0x6E	0x20	0x73	0x6F	0x64	0x00	0x74

- In Little Endian: the compiler adds **`0xF000 0000 + 1 * sizeof(int) = 0xF000 0004`**, then takes the four bytes starting from that address: 0×65 0×6E 0×20 0×73. Since this is big-endian, we combine them with the first address being the LSB, so we get 0×73206E65, which is the correct result

Endianness

- All endiannesses work as long as you're consistent. It only affects cases where you either transfer to a new endianness (different systems can use different endiannesses), or when you split/merge a block of data into smaller/larger chunks.
  - This comes up a lot when working with unions, since a union is designed to interpret the same block of data in different ways.
- Officially, C defines this to be undefined behavior; you're not supposed to do this.
  - With unions, you're supposed to use that block as only one of the components
- In practice, undefined behavior is a great way to hack someone's program, so it ends up popping up in CS 161. It also ends up showing up when you do memory dumps (see homework).

Endianness

- Big Endian is commonly used in networks (e.g. communicating data between computers).
- Little Endian is commonly used within the computer
  - The default endianness for C, RISC-V, x86, etc. In general, you'll be working with little-endian systems when programming

Summary

- Pointers, clean and compact
- Arrays, not pointers, but also not really variables
- Pointer arithmetic, how we can dynamically allocate memory
- Memory issues
- Memory endianness!

Function Pointer Example

map (actually mutate_map easier)

```
#include <stdio.h>

int x10(int), x2(int);
void mutate_map(int [], int n, int(*)(int));
void print_array(int [], int n);

int x2 (int n) {   return 2*n;   }
int x10(int n) {   return 10*n;   }

void mutate_map(int A[], int n, int(*fp)(int)) {
     for (int i = 0; i < n; i++)
          A[i] = (*fp)(A[i]);
}

void print_array(int A[], int n) {
     for (int i = 0; i < n; i++)
          printf("%d ",A[i]);
     printf("\n");
}
```

```
% ./map
3 1 4
6 2 8
60 20
80
```

```
int main(void)
{
     int A[] = {3,1,4}, n = 3;
     print_array(A, n);
     mutate_map (A, n, &x2);
     print_array(A, n);
     mutate_map (A, n, &x10);
     print_array(A, n);
}
```