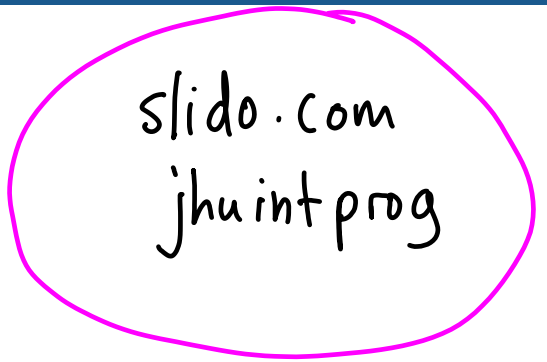


601.220 Intermediate Programming

Summer 2022, Meeting 6 (June 17th)

Today's agenda

- Exercises 9 and 10 review
- "Day 11" material
  - Dynamic memory allocation, Valgrind
  - Exercise 11
- "Day 12" material
  - Pointer arithmetic, "dynamic" 2-D arrays
  - Exercise 12

Reminders/Announcements

- HW3 due Wednesday, June 22nd
- NOW: team formation for midterm project
  - Registration form linked from Piazza post 112
    (use this post to find team members)
  - Teams of 2 or 3
  - Registration deadline: 11am on Tuesday June 21st
    (if you are not on a team by then, you will be assigned to one.)

Exercise 9 review

- Good first step in debugging a program: `break main`
  - This gives you control at the very beginning of `main`
- Use `next` (`n`) to advance to the next statement
- Use `step` (`s`) to step into a called function
  - Very important if a function is misbehaving

Exercise 9 review

To debug effectively, you need a *hypothesis* about what is going wrong.

For the `transpose` ~~function~~ program, start with the observation that the `print` function doesn't print the entire contents of the destination array.

Use `print (p)` to inspect the values of variables, array elements, etc.

Exercise 9 review

Next issue: the `transpose` function doesn't seem to correctly transpose the elements in the original array.

Step into the call to `transpose`.

Inspect "shape" and contents of the two arrays:

```
print start[0]
print start[1]
print start[2]
```

Look carefully at the code at line 13 (do tthe array subscripts make sense?)

Exercise 9 review

Debuggers are not magic.

They will not tell you what's wrong with your code. . . because they have no idea what your code is supposed to do!

They are *very* useful for seeing what your code is actually doing: they help make the internal state of the program visible.

Pro tip: learn how to set breakpoints:

- `break` *functionName*
- `break` *sourceFileName*:*lineNumber*

Use the `continue` (`c`) command to run the program until the next breakpoint is reached.

Exercise 10 review

Important application of pointers: *pass by reference* semantics for normal variables.

(Arrays are always passed by reference, but ordinary variables are passed by value by default.)

Exercise 10 review

getDate function:

```
int getDate(int *m, int *d, int *y) {
  int month, day, year;
  int rc = scanf("%d/%d/%d", &month, &day, &year);
  if (rc == 3) {
    *m = month;
    *d = day;
    *y = year;
  }
  return rc;
}
```

Exercise 10 review

`months` array:

```
const char *months[] = {
  "Jan", "Feb", "Mar", "Apr", "May", "Jun",
  "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"
};
```

Calling `getDate` from `main`:

```
printf("Enter a date: ");
while (getDate(&mon, &day, &yr) == 3) {
  printf("%s %d, %d\n", months[mon-1], day, yr);
  printf("Enter a date: ");
}
```

Exercise 10 review

Even more concise version of `getDate`:

```
int getDate(int *m, int *d, int *y) {
  return scanf("%d/%d/%d", m, d, y);
}
```

Day 11 recap questions

1. What is the difference between stack and heap memory?
2. What is dynamic memory allocation in C?
3. What is the memory leak problem?
4. What is the difference between *malloc*, *realloc*, and *calloc*?
5. What do we use valgrind to check for?
6. Consider the `exclaim` function below. Do you see any problems with this function?

1. What is the difference between stack and heap memory?

Stack memory: used for local variables, parameters, and other data required by an in-progress function call.

Key characteristic: the *lifetime* of local variables and parameters is limited to the duration of the function call for which they are allocated. (Storage for local variables, parameters, etc. is allocated automatically in a *stack frame* created when a function is called.)

Heap memory: chunks of memory can be allocated in a dedicated region of memory (the "heap").

Key characteristic: the lifetime of variables allocated in the heap is under the explicit control of the program. (I.e., the program decides when a dynamically allocated variable is no longer needed.)

2. What is dynamic memory allocation in C?

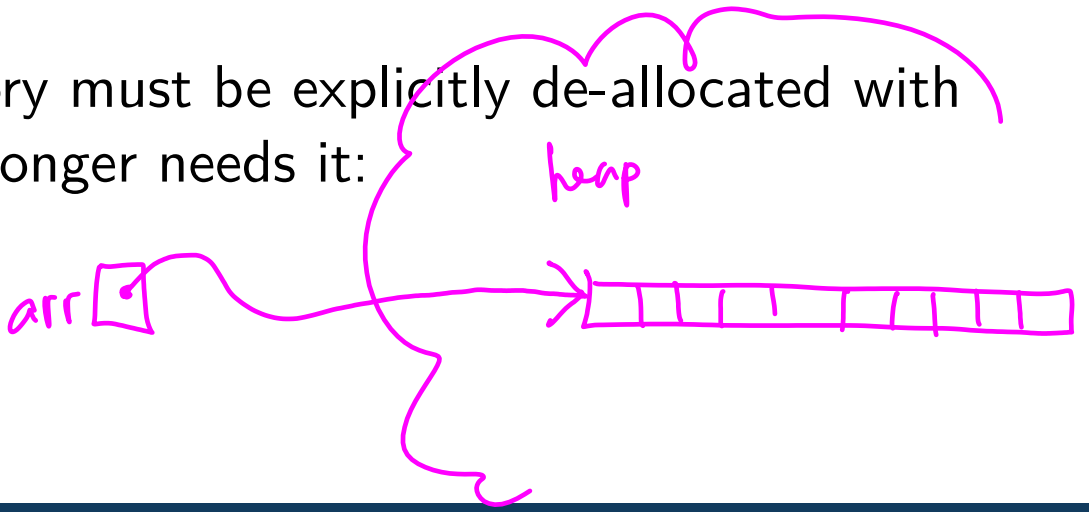
The program uses `malloc` (or `calloc`, or `realloc`) to dynamically allocate a chunk of memory of a specified size. The program can then use the chunk as a single variable, an array, etc. For example:

```
// dynamically allocate an array of 10 int elements
int *arr = (int *) malloc(10 * sizeof(int));
for (int i = 0; i < 10; i++) {
  arr[i] = (i + 1);
}
```

bytes

Dynamically allocated memory must be explicitly de-allocated with `free` when the program no longer needs it:

```
free(arr);
```



3. What is the memory leak problem?

If a program dynamically allocates memory but does not free it, it continues to exist in the heap.

The maximum amount of memory which can be allocated in the heap is finite, so a program that repeatedly allocates memory without freeing it could eventually exhaust the heap, which would cause subsequent attempts to allocate memory to fail.

Programs must take care to de-allocate dynamically allocated memory after the last use.

4. What is the difference between *malloc*, *realloc*, and *calloc*?

`malloc`: dynamically allocate block of memory of specified size.

`calloc`: like `malloc`, but contents are filled with zeroes (useful for arrays, guarantees that all elements are 0.)

`realloc`: attempt to reallocate an existing chunk of memory. Reallocation could be done "in place", or could involve allocating a new chunk of memory and copying the contents of the original block of memory.

5. What do we use valgrind to check for?

`valgrind` can check for:

1. Memory leaks (detected when the program exist)
2. Memory errors, such as

- out of bounds array accesses
- use of an uninitialized value
- access to heap memory not currently in use (e.g., dereferencing a pointer to a de-allocated block of dynamically allocated memory)

Why valgrind is useful

Testing your program regularly using `valgrind` is incredibly helpful!

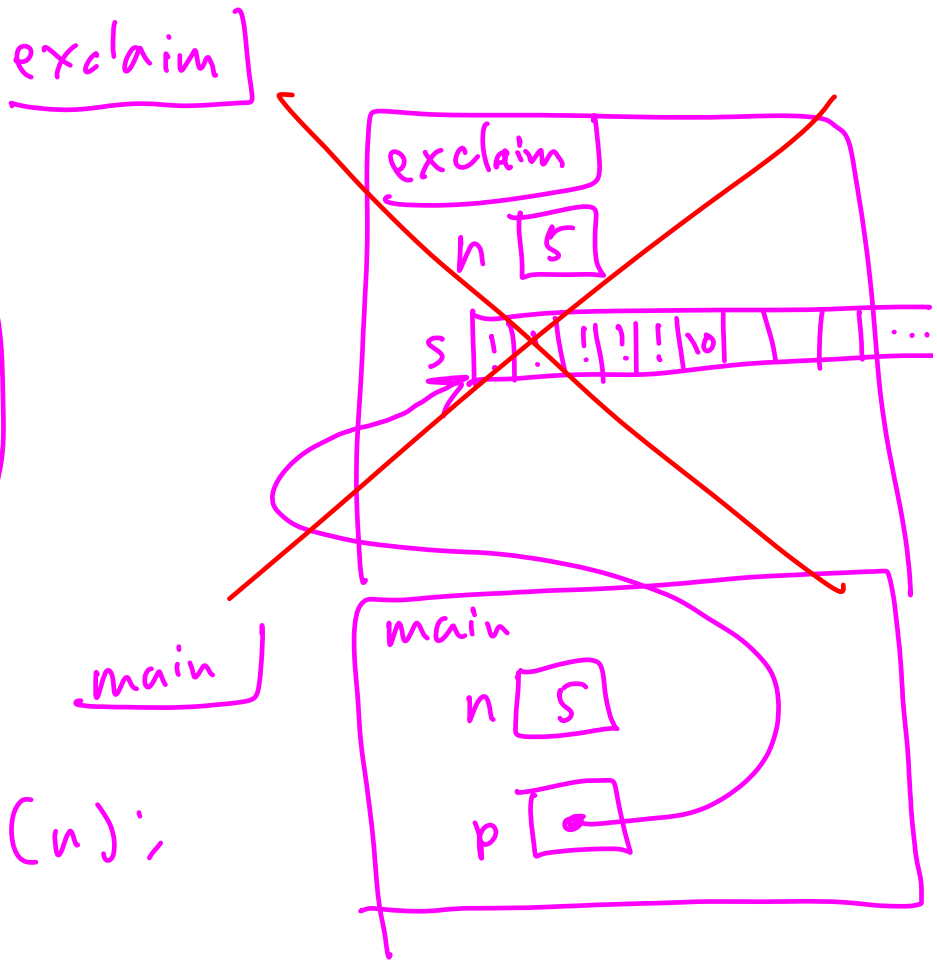
- Just because your program "works" when you run it, doesn't mean that it is free from bugs
- The kinds of bugs `valgrind` finds often lead to subtle data corruptions that can be difficult to track down by other means

Use it!

6. Consider the `exclaim` function below. Do you see any problems with this function?

The code:

```
char* exclaim(int n) {
  char s[20];
  assert(n < 20);
  for (int i = 0; i < n; i++) {
    s[i] = '!';
  }
  s[n] = '\0';
  return s;
}
```



Exercise 11

11:15

- Dynamic allocation
- Using `valgrind` to detect memory leaks and other memory errors
- Pointers to pointers
- Breakout rooms 1–10 are "social"
- Use Slack to ask for help!

Day 12 recap questions

1. What output is printed by the "Example code" below?
2. Assume that `arr` is an array of 5 `int` elements. Is the code `int *p = arr + 5;` legal?
3. Assume that `arr` is an array of 5 `int` elements. Is the code `int *p = arr + 5; printf("%d\n", *p);` legal?
4. What output is printed by the "Example code 2" below?
5. Suppose we have variables `int ra1[10] = {1, 2, 3};`, `int * ra2 = ra1;` and `int fun(int *ra);` declarations. Will `fun(ra1);` compile? Will `fun(ra2);` compile? What if we change the function declaration to `int fun(const int ra[]);`?

1. What output is printed by the "Example code" below?

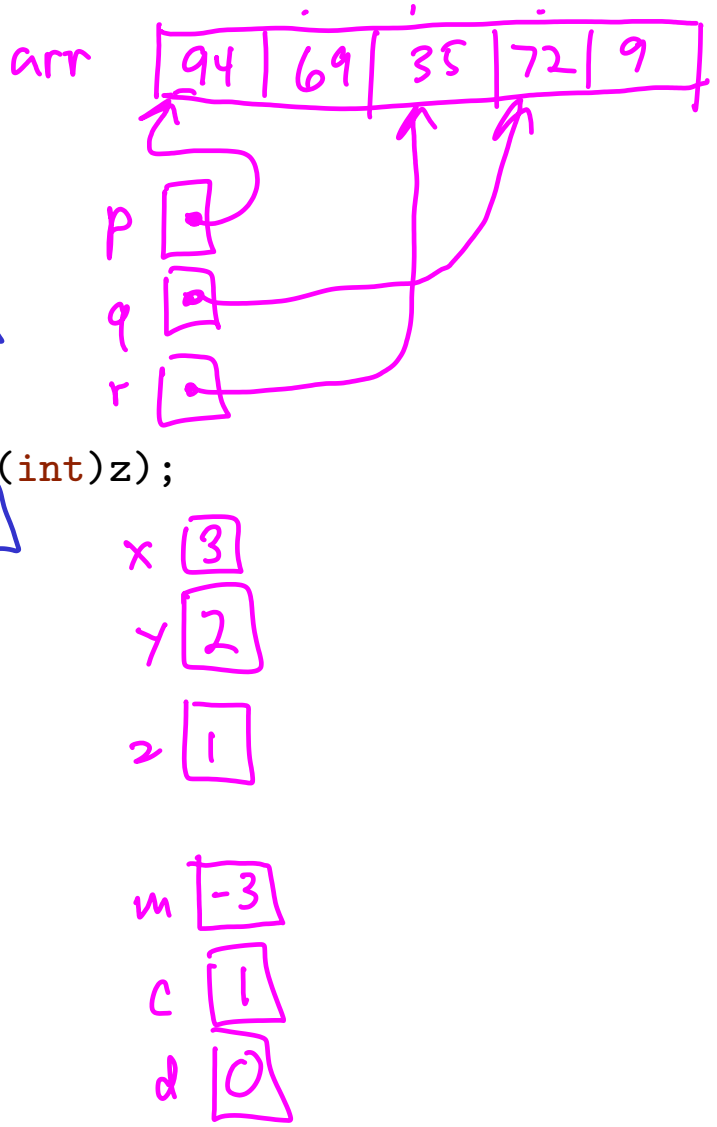
```
int arr[] = { 94, 69, 35, 72, 9 };
int *p = arr;
int *q = p + 3;
int *r = q - 1;
printf("%d %d %d\n", *p, *q, *r);
ptrdiff_t x = q - p;
ptrdiff_t y = r - p;
ptrdiff_t z = q - r;
printf("%d %d %d\n", (int)x, (int)y, (int)z);
ptrdiff_t m = p - q;
printf("%d\n", (int)m);
int c = (p < q);
int d = (q < p);
printf("%d %d\n", c, d);
```

94 72 35

3 2 1

-3

1 0



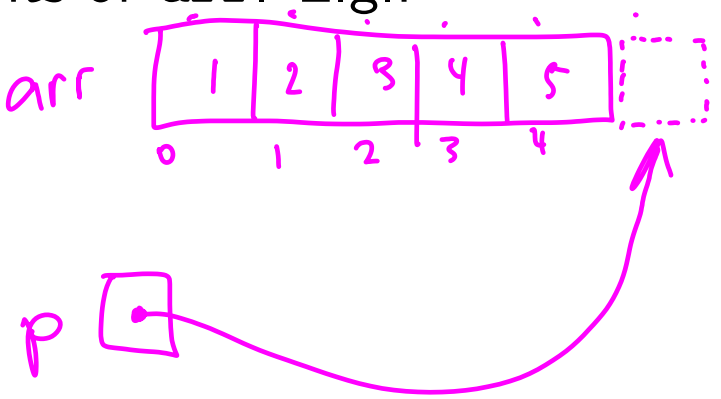
2. Assume that `arr` is an array of 5 `int` elements. Is the code `int *p = arr + 5;` legal?

Yes. It uses pointer arithmetic to compute a pointer 5 elements past the first element of `arr`.

Note that it would not be legal to *dereference* this pointer.

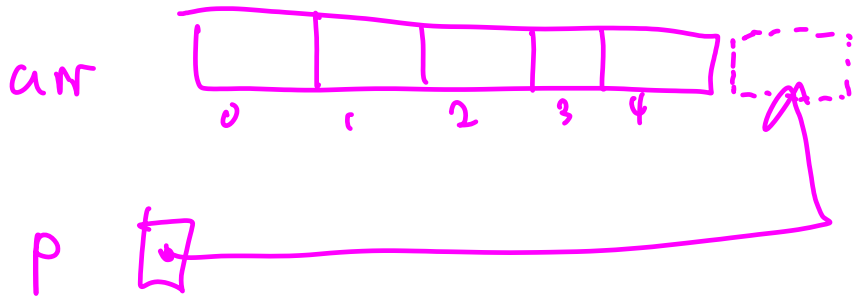
Why such a pointer might be useful: as an upper bound for a loop using a pointer to iterate through the elements of `arr`. E.g.:

```
int *p = arr + 5;
int sum = 0;
for (int *q = arr; q < p; q++) {
  sum += *q;
}
```



3. Assume that `arr` is an array of 5 `int` elements. Is the code `int *p = arr + 5; printf("%d\n", *p);` legal?

No. `p` doesn't point to a valid array element, so dereferencing it is undefined behavior.

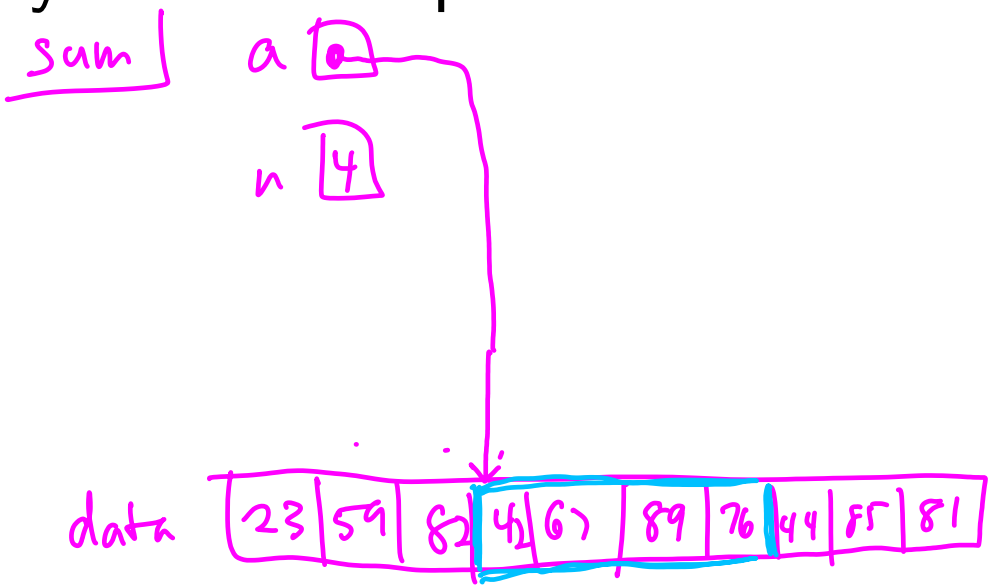


4. What output is printed by the "Example code 2" below?

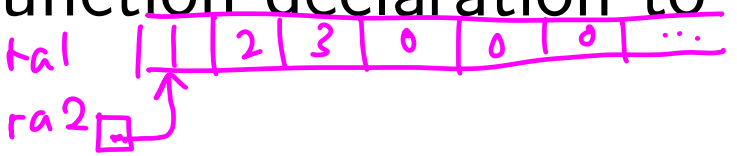
```c
#include <stdio.h>

int sum(int a[], int n) {
  int x = 0;
  for (int i = 0; i < n; i++) {
    x += a[i];
  }
  return x;
}

int main(void) {
  int data[] = { 23, 59, 82, 42, 67, 89, 76, 44, 85, 81 };
  int result = sum(data + 3, 4);
  printf("result=%d\n", result);
  return 0;
}
```



5. Suppose we have variables `int ra1[10] = {1, 2, 3};`, `int * ra2 = ra1;` and `int fun(int *ra);` declarations. Will `fun(ra1);` compile? Will `fun(ra2);` compile? What if we change the function declaration to `int fun(const int ra[]);`?



Yes, the name of an array of `int` elements will "decay" into a pointer to the first element of the array if used without the subscript operator.

Yes, `ra2` is a poiner to `int`, which is the type of argument expected by `fun`.

Yes, a pointer to `int` can be passed to a function expecting pointer to `const int`. (Note that it's *not* allowed to pass a pointer to `const int` to a function expecting a pointer to (non-`const`) `int`.)

Exercise 12

- Using pointer arithmetic to treat regions of arrays as "sub-arrays"
- Using pointer difference to translate a pointer to an element into the element's index (by subtracting the "base pointer", i.e., the pointer to the first element)
- Breakout rooms 1–10 are "social"
- Use Slack to ask for help!

Notes

```
int a[10] = {1, 2, 3}; // all elts init'ed

int b[10]; // no elt init'ed
```

Notes

Notes

Notes

Notes