601.220 Intermediate Programming

Spring 2023, Day 12 (February 17th)

Today's agenda

- Exercise 11 review
- Pointer arithmetic, "dynamic" 2-D arrays
- Exercise 12

Reminders/Announcements

- HW2 due **this evening** by 11 pm
  - Written assignment, late submissions not accepted
- HW3 due Friday, Feb 24th

Exercise 11 review

`pairwise_sum.c`: When running the program using valgrind:

```
valgrind --leak-check=full ./pairwise_sum
```

A memory leak is reported:

```
==17736== 16 bytes in 1 blocks are definitely lost in loss record 1 of 1
==17736==    at 0x483B7F3: malloc (in /usr/lib/x86_64-linux-gnu/valgrind/vgprel
==17736==    by 0x10922B: pairwise_sum (pairwise_sum.c:28)
==17736==    by 0x109399: main (pairwise_sum.c:57)
```

`valgrind` indicates there is a memory leak: the memory is allocated in `pairwise_sum.c` at line 28

Exercise 11 review

In the code:

```
int *pairsum2 = pairwise_sum(pairwise_sum(array, 5), 4);
// ...
free(pairsum2);
```

Issue: `pairwise_sum` returns a pointer to a dynamically allocated array, but for the "inner" call, the array is never freed.

Fix:

```
int *a = pairwise_sum(array, 5);
int *pairsum2 = pairwise_sum(a, 4);
// ...
free(pairsum2);
free(a);
```

Exercise 11 review

`primes.c`:

Issue: the `set_primes` function needs to call `realloc` if the array of results needs to be increased in size.

However, `realloc` can and usually does return a pointer to a new dynamic array (with a different memory address).

Unless `set_primes` can modify the list pointer in main, the main function has no way of knowing the address of the re-allocated array.

Exercise 11 review

Sketch showing the problem with the original code:

Exercise 11 review

Solution: change `set_primes` so that it takes a pointer to the `list` pointer variable in the main function.

```
// set_primes function: originally
int set_primes( int *list , int capacity )

// updated
int set_primes( int **list , int capacity )

// in main function
int *list = /* initial allocation of array */

// original call to set_primes
int prime_count = set_primes( list , capacity );

// updated
int prime_count = set_primes( &list , capacity );
```

Exercise 11 review

Sketch showing how having `set_primes` take a pointer to a pointer solves the problem:

Exercise 11 review

Changes to `set_primes`: essentially, everywhere that `list` was mentioned, we now want `*list` so that we are referring (indirectly) to the `list` pointer variable in main.

One issue: array subscript operator has higher precedence than the pointer dereference operator (`*`)

So, instead of changing

```
list[idx++] = n;
```

to

```
*list[idx++] = n;
```

it should be

```
(*list)[idx++] = n;
```

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);
```

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 poiner 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)
- Talk to us if you have questions!

Notes

Notes

Notes

Notes

Notes