

# 601.220 Intermediate Programming

Summer 2022, Meeting 7 (June 22nd)

# Today's agenda

- Exercises 11 and 12 review
- “Day 13” material
  - Lifetime/scope, struct types, random number generation
  - Exercise 13
- “Day 14” material
  - Binary file I/O, bitwise operations
  - Exercise 14

# Reminders/Announcements

- HW3 due *today*
- Midterm project teams
  - If you were not registered on a team as of 11pm last night, you will be assigned to a team today
  - Team repositories will be available by end of day today
- Midterm project: overview in class on Friday June 24th, due Friday, July 1st

## 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/vgpreload_memcheck-amd64-linux.so)
==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

## 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;
```

## Exercise 12 review

Declaration of search function:

How it is called:

```
pos = search(arr1, arr1 + 10, 318);
```

Declaration:

```
int *search(int *start, int *end, int searchval);
```

## Exercise 12 review

Useful property when lower bound of search range is inclusive, and upper bound is exclusive:  $\text{end} - \text{start}$  is the number of elements in the range. So:

```
int *search(int *start, int *end, int searchval) {  
    int num_elts = (int) (end - start);  
    if (num_elts < 1) {  
        return NULL; // no elements in range  
    } else {  
        // general case: check middle element, if it's equal to  
        // searchval, success, otherwise continue recursively on  
        // left or right side of range  
    }  
}
```

## Exercise 12 review

```
// search, general case
int *mid = start + (num_elts/2);
if (*mid == searchval) {
    return mid; // success, found the search value
} else if (*mid < searchval) {
    // continue recursively in right side of range
} else {
    // continue recursively in left side of range
}
```

## Exercise 12 review

```
// in the test code, finding the index of the matching element
pos = search(arr1, arr1 + 10, 318);
assert(pos != NULL);
assert(*pos == 318);
// TODO: compute the index of the matching element
index = pos - arr1; // <-- add this
assert(2 == index);
```

## Exercise 12 review

General observation about 2-D arrays: if  $p$  is a pointer to an element, and  $N$  is the number of columns in one row, then

$p + N$

yields a pointer to an element that is in the same column and next row from the element  $p$  points to. Picture:

## Exercise 12 review

makeCol:

```
// TODO: declare the unit variable (array of 9 integers, to be returned)  
int *unit = malloc(9 * sizeof(int));
```



## Exercise 12 review

makeCube:

```
// TODO: declare the unit variable (array of 9 integers, to be returned  
int *unit = malloc(9 * sizeof(int));
```

## Exercise 12 review

checkRows:

```
// TODO: call check on current row and add to variable good  
good += check(&table[r][0]);
```

Observation: elements in a single row are contiguous in memory  
(each row of a 2-D array can be treated as a 1-D array).

## Exercise 12 review

checkCols:

```
for (int c = 0; c < SIZE; c++) {  
    // TODO: call makeCol on current column and assign result to column  
    column = makeCol(&table[0][c]);    // <-- get one column of values  
    good += check(column);  
    free(column);                      // <-- free dynamic array  
}
```

## Exercise 12 review

checkCubes:

```
// TODO: call makeCube on current cube and assign result to variable cube  
cube = makeCube(&table[r][c]);    // <-- get 3x3 "cube" of values  
good += check(cube);  
free(cube);                       // <-- free dynamic array
```

## Exercise 12 review

`main` (in `sudoku.c`): code does not call `fclose` to close input file: should modify `main` function so that `infile` is guaranteed to be closed (using `fclose`) if it is opened successfully.

Makefile: `CFLAGS` should include the `-g` option (to enable debug symbols).

Running `valgrind`:

```
valgrind ./main --leak-check=full --show-leak-kinds=all <name of input file>
```

## Day 13 recap questions

- ① What is *struct* in C?
- ② How are the fields of a struct passed into a function - by value or by reference?
- ③ What is the size of a *struct*? What is structure padding in C?
- ④ What is the difference between lifetime and scope of a variable?
- ⑤ What is variable shadowing (i.e. hiding)?
- ⑥ What is the output of the below program?

# 1. What is *struct* in C?

`struct` introduces a *used-defined data type*.

Very much like a class in Java or Python, but with only the ability to include member variables, not member functions.

An instance of a struct is a “bundle” of variables that are packaged as a single entity.

Example:

```
struct Point {  
    int x, y;  
};
```

```
// ... elsewhere in the program ...  
struct Point p = { .x = 2, .y = 3 };
```

## 2. How are the fields of a struct passed into a function - by value or by reference?

Instances of a struct type are passed by value. E.g.

```
struct Point { int x, y; };

void f(struct Point p, int dx) {
    p.x += dx;
}

int main(void) {
    struct Point q = { .x = 4, .y = 5 };
    f(q);
    printf("%d,%d\n", q.x, q.y); // prints "4,5"
    return 0;
}
```



### 3. What is the size of a *struct*? What is structure padding in C?

`sizeof(struct Foo)` is the sum of the sizes of the fields of struct Foo, plus the total size of any padding inserted by the compiler to ensure that fields are correctly aligned.

*alignment*: the memory address of a variable (including a field variable in an instance of a struct type) must be a multiple of the size of the field.

E.g., a 4-byte `int` variable (or struct field) must have its storage allocated starting at a machine address that is a multiple of 4.

The compiler will insert padding automatically: you don't need to do anything special. `sizeof(struct Foo)` will always take the padding into account. Just trust that the compiler will figure out the right struct layout to use.

## struct padding example

```
struct Foo {  
    char a;  
    int b;  
    long c;  
};  
  
// ...  
  
struct Foo f;  
printf("%lu\n", sizeof(f));
```

## 4. What is the difference between lifetime and scope of a variable?

*Lifetime*: the interval from (1) the point in time when a variable is created, to (2) the point in time when a variable is destroyed. Examples:

- the lifetime of a local variable is the duration of the function call
- the lifetime of a global variable is the duration of the entire program

*Scope*: the region of the program code in which a variable may be accessed. Examples:

- the scope of a local variable is from its declaration to the closing “}” of the block in which it’s defined
- the scope of a global variable is the entire program (assuming that there is a declaration or definition of the variable in the current block, or in the enclosing block)

## 5. What is variable shadowing (i.e. hiding)?

Shadowing: a variable declaration in a nested scope has the same name as a variable in an “outer” scope.

# Shadowing example

```
int x;

void foo(int x) {
    {
        int x = 5;
        printf("%d\n", x); // prints "5"
    }
    printf("%d\n", x); // prints "4"
}

int main(void) {
    x = 3;
    foo(4);
    printf("%d\n", x); // prints "3"
    return 0;
}
```

## 6. What is the output of the below program?

```
#include <stdio.h>
int foo;
void bar() {
    int foo = 3;
    {
        extern int foo;
        printf("%d; ", foo);
        foo = 2;
    }
    printf("%d; ", foo);
}
void baz() { printf("%d; ", foo); }
int main() {
    {
        int foo = 5;
        bar();
        printf("%d; ", foo);
    }
    baz();
    return 0;
}
```

## . VS. ->

To access a member variable of a struct instance directly, use the “.” operator. To access a member variable of a struct instance indirectly via a pointer, use the -> operator.

Note that `p->x` means exactly the same thing as `(*p)->x`. It's just a more convenient syntax.

## Example of . vs. ->

```
struct Player { int x; int y; int health; };
```

```
struct Player player;
```

```
player.x = 42;
```

```
player.y = 17;
```

```
struct Player *p = &player;
```

```
p->health = 100;
```



## Exercise 13

- Working with struct types, including pointers to instances of struct types
- Breakout rooms 1–10 are “social”
- Use Slack to let us know if you have any questions!

## Day 14 recap questions

- ① How do we read/write binary files in C?
- ② What character represents the bitwise XOR operation? How does it differ from the OR operation?
- ③ What happens if you apply the bitwise operation on an integer value? (extra: what if we apply to floats)
- ④ What is the result of `(15 >> 2) || 7`?
- ⑤ What is the result of `(15 >> 2) | 7`?

# 1. How do we read/write binary files in C?

Use "rb" or "wb" when calling fopen, and use fread or fwrite to read binary data value(s).

E.g., read array of 20 int values from a file of binary data:

```
FILE *in = fopen("input.dat", "rb");
if (in == NULL) { /* handle error */ }
else {
    int *arr = malloc(sizeof(int) * 20);
    int rc = fread(arr, sizeof(int), 20, in);
    if (rc != 20) { /* handle error */ }
}
// arr now points to array of 20 elements read from input file
```

*Warning:* this code is not portable across CPU architectures due to byte ordering. For multi-byte data values some CPUs store least significant byte first ("little endian"), some CPUs store most significant byte first ("big endian")

# Binary data representation

“Binary” means “base 2”.

Digital computers represent all numbers using base-2 rather than base 10. For example:

42 in base 10:  $4 \times 10^1 + 2 \times 10^0$

42 in base 2:

$$\begin{aligned} &1 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 \\ &= 1 \times 32 + 0 \times 16 + 1 \times 8 + 0 \times 4 + 1 \times 2 + 0 \times 1 \end{aligned}$$

42 as a sequence of binary digits (“bits”): 101010

All data values are represented in binary (base-2) at the machine level. “Bitwise” operators allow you to work directly with binary values.

## 2. What character represents the bitwise XOR operation? How does it differ from the OR operation?

Bitwise OR: combine two binary values by computing the result of the OR operation on each pair of binary digits.

Operator: `|`

Logic:  $0|0=0$ ,  $0|1=1$ ,  $1|0=1$ ,  $1|1=1$

Bitwise XOR: combine two binary values by computing the result of the XOR (exclusive or) operation on each pair of binary digits.

Operator: `^`

Logic:  $0^0=0$ ,  $0^1=1$ ,  $1^0=1$ ,  $1^1=0$

# Bitwise AND

Bitwise AND: combine two binary values by computing the result of the AND operation on each pair of binary digits.

Operator: `&`

Logic:  $0 \& 0 = 0$ ,  $0 \& 1 = 0$ ,  $1 \& 0 = 0$ ,  $1 \& 1 = 1$

### 3. What happens if you apply the bitwise operation on an integer value? (extra: what if we apply to floats)

The two-operand bitwise operators (`|`, `^`, `&`) perform a logical operation (OR, XOR, AND) on each pair of bits in the two operands.

The operands must be values belonging to an “integral” (integer-like) type.

E.g., `int`, `unsigned`, `long`, `unsigned long`, `char`, `unsigned char`, etc.

Bitwise operations may *not* be performed on floating point (`float` or `double`) values.

#### 4. What is the result of `(15 >> 2) || 7`?

15 in base-2 is 1111

“>>” is the right shift operator, shifting 1111 two bits to the right yields 0011, which is equal to 3.

Any non-zero integer is considered TRUE, so 3 is true.

If the left operand of `||` is true, the entire expression is true (and the right operand is not evaluated.) All logical and relational operators yield 0 when false and 1 when true.

So, the result of `(15 >> 2) || 7` is 1.



5. What is the result of  $(15 \gg 2) \mid 7$ ?

$\mid$  is the bitwise OR operator

15 in binary is 1111

$(15 \gg 2)$  is 0011

7 in binary is 0111

```
  0011
| 0111
-----
```

0111 -- bit is 1 where either operand has a 1 bit  
which is equal to 7

## Exercise 14

- arrays and strings
- bitwise operations
- Breakout rooms 1–10 are “social”
- Use Slack to let us know if you have any questions!

# Notes

# Notes

# Notes

# Notes

# Notes