CMPUT201W20B2 Week 7

Abram Hindle

March 10, 2020

Contents

1 Week7

1.1 Copyright Statement

If you are in CMPUT201 at UAlberta this code is released in the public domain to you.

Otherwise it is (c) 2020 Abram Hindle, Hazel Campbell AGPL3.0+

1.1.1 License

Week 3 notes Copyright (C) 2020 Abram Hindle, Hazel Campbell

This program is free software: you can redistribute it and/or modify it under the terms of the GNU Affero General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Affero General Public License for more details.

You should have received a copy of the GNU Affero General Public License along with this program. If not, see `https://www.gnu.org/licenses/`.

1.1.2 Hazel Code is licensed under AGPL3.0+

Hazel's code is also found here `https://github.com/hazelybell/examples/tree/C-2020-01`

Hazel code is licensed: The example code is licensed under the AGPL3+ license, unless otherwise noted.

1.2 Init ORG-MODE

```
;; I need this for org-mode to work well
;; If we have a new org-mode use ob-shell
;; otherwise use ob-sh --- but not both!
```

```
(if (require 'ob-shell nil 'noerror)
  (progn
    (org-babel-do-load-languages 'org-babel-load-languages '((shell . t))))
  (progn
    (require 'ob-sh)
    (org-babel-do-load-languages 'org-babel-load-languages '((sh . t)))))
(org-babel-do-load-languages 'org-babel-load-languages '((C . t)))
(org-babel-do-load-languages 'org-babel-load-languages '((python . t)))
(setq org-src-fontify-natively t)
(setq org-confirm-babel-evaluate nil) ;; danger!
(custom-set-faces
 '(org-block ((t (:inherit shadow :foreground "black")))))
```

1.2.1 Org export

```
(org-html-export-to-html)
(org-latex-export-to-pdf)
(org-ascii-export-to-ascii)
```

1.3 Org Template

Copy and paste this to demo C

```
#include <stdio.h>

int main(int argc, char**argv) {
    return 0;
}
```

1.4 Remember how to compile?

gcc -std=c99 -Wall -pedantic -Werror -o programname programname.c

1.5 Enums!

Enums are enumerations, which is just a convienant way to make symbols that have different values of the same type. Enums allow us to read and write values from files and inputs and extract their symbolic meaning.

Enums are fundamental to symbolic computation.

Enum work good for switch cases, if statements, for loops.

Enums are good for representing the type of something or a category.

1.5.1 Enum Example

Enums are good for representing states, symbols, simple values, etc.

```
#include <stdio.h>
#include <stdlib.h>

#define N_DIRECTIONS 4
enum direction {
    UP=1, DOWN=2, LEFT=3, RIGHT=0
};
typedef enum direction Direction;

const char * const direction_names[N_DIRECTIONS] = {
    [UP] = "Up",
    [DOWN] = "Down",
    [LEFT] = "Left",
    [RIGHT] = "Right"
};

Direction clockwise(Direction direction) {
    switch (direction) {
        case UP:
            return RIGHT;
        case RIGHT:
            return DOWN;
        case DOWN:
            return LEFT;
        case LEFT:
            return UP;
        default:
            abort();
    }
}

int main() {
    Direction d = UP;
    for (int i = 0 ; i < 10; i++) {
        d = clockwise(d);
        printf("%d %s\t[%d]\n", i, direction_names[d], d);
    }
```

```
}
```

```
0 Right [0]
1 Down [2]
2 Left [3]
3 Up [1]
4 Right [0]
5 Down [2]
6 Left [3]
7 Up [1]
8 Right [0]
9 Down [2]
```

1.5.2 $\text{enum}_{\text{typedef.c}}$

Enums are annoying to type. Typing enum enumname all the time is repetitive. Typedefs allow us to label enum types with 1 word.

Typedef this
enum enumname { ... } ;
with:
typedef enum enunumae Enumename ;

```
#include <stdio.h>
#include <stdlib.h>

enum flavor {
    VANILLA,
    CHOCOLATE,
    STRAWBERRY,
};

typedef enum flavor Flavor;

int main() {
    Flavor favourite = VANILLA;
    printf("favourite=%d\n", favourite);
    printf("sizeof(favourite)=%zu\n",
           sizeof(favourite));

    switch (favourite) {
```

```
        case VANILLA:
            printf("favourite=VANILLA\n");
            break;
        case CHOCOLATE:
            printf("favourite=CHOCOLATE\n");
            break;
        case STRAWBERRY:
            printf("favourite=STRAWBERRY\n");
            break;
        default:
            abort();
    }
}

favourite=0
sizeof(favourite)=4
favourite=VANILLA
```

1.5.3 EnumStart

```
#include <stdio.h>
#include <stdlib.h>

enum flavor {
    VANILLA = 100,
    CHOCOLATE,
    STRAWBERRY,
};

typedef enum flavor Flavor;

const char * flavorString(Flavor flavor) {
    switch (flavor) {
        case VANILLA:
            return "Vanilla";
        case CHOCOLATE:
            return "Chocolate";
        default:
            abort();
    }
```

```
}


int main() {
    printf("VANILLA=%d\n", VANILLA);
    printf("CHOCOLATE=%d\n", CHOCOLATE);
    printf("STRAWBERRY=%d\n", STRAWBERRY);
    printf("sizeof(Flavor)=%zu\n",
           sizeof(Flavor));
    puts(flavorString(VANILLA));
    puts(flavorString(100));

}
```

```
VANILLA=100
CHOCOLATE=101
STRAWBERRY=102
sizeof(Flavor)=4
Vanilla
Vanilla
```

1.5.4 Enumassign

```
#include <stdio.h>
#include <stdlib.h>

enum flavor {
    VANILLA = 100,
    CHOCOLATE = 200,
    STRAWBERRY = 300,
};

typedef enum flavor Flavor;

int main() {
    printf("VANILLA=%d\n", VANILLA);
    printf("CHOCOLATE=%d\n", CHOCOLATE);
    printf("STRAWBERRY=%d\n", STRAWBERRY);
    printf("sizeof(Flavor)=%zu\n",
           sizeof(Flavor));
```

```
}
```

```
VANILLA=100
CHOCOLATE=200
STRAWBERRY=300
sizeof(Flavor)=4
```

1.5.5 Enum$_{\text{looptrick.c}}$

This is a fun trick to set a maximum value for your enum by using another symbol

```
#include <stdio.h>
#include <stdlib.h>

// this only works as long as we don't provide our
// own values!

enum flavor {
    VANILLA,
    RHUBARB,
    CHOCOLATE,
    STRAWBERRY,
    N_FLAVORS // Get the free max enum here
};

typedef enum flavor Flavor;

int main() {
    printf("VANILLA=%d\n", VANILLA);
    printf("CHOCOLATE=%d\n", CHOCOLATE);
    printf("STRAWBERRY=%d\n", STRAWBERRY);
    printf("N_FLAVORS=%d\n", N_FLAVORS);
    printf("sizeof(Flavor)=%zu\n",
           sizeof(Flavor));

    for (Flavor flavor = 0; flavor < N_FLAVORS; flavor++) {
        switch (flavor) {
            case VANILLA:
                printf("flavor=VANILLA\n");
                break;
```

```
            case CHOCOLATE:
                printf("flavor=CHOCOLATE\n");
                break;
            case STRAWBERRY:
                printf("flavor=STRAWBERRY\n");
                break;
            case RHUBARB:
                printf("flavor=RHUBARB\n");
                break;
            default:
                abort();
        }
    }
}
```

```
VANILLA=0
CHOCOLATE=2
STRAWBERRY=3
N_FLAVORS=4
sizeof(Flavor)=4
flavor=VANILLA
flavor=RHUBARB
flavor=CHOCOLATE
flavor=STRAWBERRY
```

1.5.6 Enum Int

Enum are just integers. And you can treat them as such.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

enum flavor {
    VANILLA,
    CHOCOLATE,
    STRAWBERRY,
    N_FLAVORS // LOOK MA! No Defines! Cute trick, might surprise people.
};

typedef enum flavor Flavor;
```

```
// Here we use the fact that enums are really just ints!
Flavor random_flavor() {
    return (rand() % N_FLAVORS);
}

void check_flavor(Flavor flavor) {
    if (flavor >= N_FLAVORS) {
        abort();
    }
    // Since a flavor is just an int, it could be negative...
    if (flavor < 0) {
        abort();
    }
}

const char * get_flavor_name(Flavor flavor) {
    check_flavor(flavor);
    // Here we use "Designated Initializers"!
    const char * const flavor_names[N_FLAVORS] = {
        [CHOCOLATE] = "Hamburger flavor",
        [VANILLA] = "Raspberry",
        [STRAWBERRY] = "Those packets that come in the ramen"
    };
    const char * flavor_name = flavor_names[flavor];
    return flavor_name;
}

int main() {
    srand(time(NULL));
    for (int i = 0 ; i < 4; i++) {
       Flavor flavor = random_flavor();
       printf(
           "flavor %d = %s\n",
           flavor,
           get_flavor_name(flavor)
       );
    }
}
```

```
flavor 1 = Hamburger flavor
flavor 2 = Those packets that come in the ramen
flavor 2 = Those packets that come in the ramen
flavor 2 = Those packets that come in the ramen
```

1.5.7 Another motivating ENUM Example

```
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

enum card_face {
    ACE = 1,
    FACE2,
    FACE3,
    FACE4,
    FACE5,
    FACE6,
    FACE7,
    FACE8,
    FACE9,
    FACE10,
    JACK,
    QUEEN,
    KING,
};

typedef enum card_face CardFace;

enum card_suit {
    CLUBS,
    HEARTS,
    DIAMONDS,
    SPADES
};

typedef enum card_suit CardSuit;

struct playing_card {
```

```
    CardFace face;
    CardSuit suit;
};

typedef struct playing_card PlayingCard;

#define HANDSIZE 5

// A flush is a hand where all cards have the same suit
// like 5 diamonds or 5 hearts
bool isFlush(PlayingCard hand[HANDSIZE]) {
    CardSuit suit = hand[0].suit;
    for (int i = 1;  i < HANDSIZE; i++ ) {
        if (suit != hand[i].suit) {
            return false;
        }
    }
    return true;
}

int main() {
    srand(time(NULL));
    PlayingCard hand[5] = {
        {ACE, CLUBS},
        {FACE2, CLUBS},
        {KING, CLUBS},
        {JACK, CLUBS},
        {FACE4, CLUBS}
    };
    printf("isFlush? %u\n", isFlush(hand));
    PlayingCard secondHand[5] = {
        {ACE, CLUBS},
        {ACE, SPADES},
        {ACE, HEARTS},
        {ACE, DIAMONDS},
        {ACE, CLUBS} // CHEATER
    };
    printf("isFlush? %u\n", isFlush(secondHand));

}
```

```
isFlush? 1
isFlush? 0
```

1.6 Unions

- Unions are a way to all the same types to share the same memory.
- Some types like unsigned integers have different sizes:

```
#include <stdio.h>
int main() {
    printf("sizeof(unsigned char):\t\t%ld\n",
        sizeof(unsigned char));
    printf("sizeof(unsigned short):\t\t%ld\n",
        sizeof(unsigned short));
    printf("sizeof(unsigned int):\t\t%ld\n",
        sizeof(unsigned int));
    printf("sizeof(unsigned long):\t\t%ld\n",
        sizeof(unsigned long));
    printf("sizeof(unsigned long long):\t%ld\n",
        sizeof(unsigned long long));
    printf("sizeof(unsigned long long int):\t%ld\n",
        sizeof(unsigned long long int));

}
```

```
sizeof(unsigned char):      1
sizeof(unsigned short):     2
sizeof(unsigned int):       4
sizeof(unsigned long):      8
sizeof(unsigned long long):     8
sizeof(unsigned long long int):     8
```

```
#include <stdio.h>
/*
[C][S][I][I][L][L][L][L]
 S  I  L  L
 I  L
 L
*/
union uints {
```

```
    unsigned char a_char;
    unsigned short a_short;
    unsigned int an_int;
    unsigned long a_long;
};
typedef union uints UInts;

#define EXAMPLES 7
int main() {
    unsigned long longs[EXAMPLES] =
        { 0, 1000, 1000000, 10000000000,
          65535, 4294967295, 18446744073709551615UL };
    UInts uints;
    for (int i = 0 ; i < EXAMPLES; i++) {
        uints.a_long = longs[i];
        printf("For the long %lu:\n", longs[i]);
        printf("\tchar\t %hhu\n", uints.a_char);
        printf("\tshort\t %hu\n", uints.a_short);
        printf("\tint\t %u\n", uints.an_int);
        printf("\tlong\t %lu\n", uints.a_long);
    }
    // lets see if we can overflow
    uints.a_char++;
    printf("Overflow kept local\t %lu\n", uints.a_long);
}

For the long 0:
char  0
short  0
int  0
long  0
For the long 1000:
char  232
short  1000
int  1000
long  1000
For the long 1000000:
char  64
short  16960
int  1000000
```

```
long  1000000
For the long 10000000000:
char  0
short  58368
int  1410065408
long  10000000000
For the long 65535:
char  255
short  65535
int  65535
long  65535
For the long 4294967295:
char  255
short  65535
int  4294967295
long  4294967295
For the long 18446744073709551615:
char  255
short  65535
int  4294967295
long  18446744073709551615
Overflow kept local  18446744073709551360
```

1.6.1 Structs versus Unions

```
#include <stdio.h>

struct uints {
    unsigned char a_char;
    unsigned short a_short;
    unsigned int an_int;
    unsigned long a_long;
};
typedef struct uints UInts;

int main() {
    UInts uints;
    uints.a_long = 0;
    printf("Hi I'm a struct!\n");
    printf("sizeof(uints)=%zu\n", sizeof(uints));
```

```
    printf("sizeof(uints.a_char)=%zu\n", sizeof(uints.a_char));
    printf("sizeof(uints.a_short)=%zu\n", sizeof(uints.a_short));
    printf("sizeof(uints.an_int)=%zu\n", sizeof(uints.an_int));
    printf("sizeof(uints.a_long)=%zu\n", sizeof(uints.a_long));
    printf("&uints=        %p\n", (void *) &uints);
    printf("&uints.a_char= %p\n", (void *) &(uints.a_char));
    printf("&uints.a_short=%p\n", (void *) &(uints.a_short));
    printf("&uints.an_int= %p\n", (void *) &(uints.an_int));
    printf("&uints.a_long= %p\n", (void *) &(uints.a_long));
}

Hi I'm a struct!
sizeof(uints)=16
sizeof(uints.a_char)=1
sizeof(uints.a_short)=2
sizeof(uints.an_int)=4
sizeof(uints.a_long)=8
&uints=        0x7ffddf5be350
&uints.a_char= 0x7ffddf5be350
&uints.a_short=0x7ffddf5be352
&uints.an_int= 0x7ffddf5be354
&uints.a_long= 0x7ffddf5be358

#include <stdio.h>

union uints {
    unsigned char a_char;
    unsigned short a_short;
    unsigned int an_int;
    unsigned long a_long;
};
typedef union uints UInts;

int main() {
    UInts uints;
    uints.a_long = 0;
    printf("Hi I'm a Union!\n");
    printf("sizeof(uints)=%zu\n", sizeof(uints));
    printf("sizeof(uints.a_char)=%zu\n", sizeof(uints.a_char));
    printf("sizeof(uints.a_short)=%zu\n", sizeof(uints.a_short));
```

```
    printf("sizeof(uints.an_int)=%zu\n", sizeof(uints.an_int));
    printf("sizeof(uints.a_long)=%zu\n", sizeof(uints.a_long));
    printf("&uints=        %p\n", (void *) &uints);
    printf("&uints.a_char= %p\n", (void *) &(uints.a_char));
    printf("&uints.a_short=%p\n", (void *) &(uints.a_short));
    printf("&uints.an_int= %p\n", (void *) &(uints.an_int));
    printf("&uints.a_long= %p\n", (void *) &(uints.a_long));
}
```

```
Hi I'm a Union!
sizeof(uints)=8
sizeof(uints.a_char)=1
sizeof(uints.a_short)=2
sizeof(uints.an_int)=4
sizeof(uints.a_long)=8
&uints=        0x7ffe14f8b080
&uints.a_char= 0x7ffe14f8b080
&uints.a_short=0x7ffe14f8b080
&uints.an_int= 0x7ffe14f8b080
&uints.a_long= 0x7ffe14f8b080
```

1.6.2 Union considerations

- they are aligned at the starting byte of each member.
- overflows are kept local to the member being addressed

1.6.3 Type Punning

- Type punning is breaking the type system to achieve a goal
- in C it is undefined behaviour to write to 1 part of the union and then read from that data using a different overlapping member. Yet it pretty common practice.
- GCC and others typically allow it.

```
#include <stdio.h>
#include <stdint.h>
#include <limits.h>
#include <stdlib.h>
```

```
/* This is super useful, but we can't do it in C99 */

struct multi_type {
    enum {
        NOTHING,
        AN_INT,
        A_FLOAT
    } which;
    union {
        int32_t an_int;
        float a_float;
    };
};
typedef struct multi_type MultiType;

void print_mt(MultiType mt) {
    if (mt.which == NOTHING) {
        printf("nothing");
    } else if (mt.which == AN_INT) {
        printf("%d", (int) mt.an_int);
    } else if (mt.which == A_FLOAT) {
        printf("%e", mt.a_float);
    } else {
        abort();
    }
}

void print_mt_array(MultiType *mt_array, size_t length) {
    size_t idx;
    for (idx = 0; idx < length; idx++) {
        print_mt(mt_array[idx]);
        printf(" ");
    }
    printf("\n");
}

MultiType new_mt_int(int value) {
    MultiType new;
    new.which = AN_INT;
    new.an_int = value;
```

```
    return new;
}

MultiType new_mt_float(float value) {
    MultiType new;
    new.which = A_FLOAT;
    new.a_float = value;
    return new;
}

int main() {
    MultiType mt_array[4] = { { NOTHING } };
    mt_array[0] = new_mt_int(24);
    mt_array[1] = new_mt_int(48);
    mt_array[2] = new_mt_float(0.24);
    mt_array[3] = new_mt_float(0.12);
    printf("\n");
    print_mt_array(mt_array, 4);
}
```

The error message:
/tmp/babel-27627ARt/C-src-27627FnU.c:24:6: error: ISO C99 doesn't support unnamed structs/unions [-Werror=pedantic] }; ^ cc1: all warnings being treated as errors /bin/bash: /tmp/babel-27627ARt/C-bin-27627Sxa: Permission denied

1.6.4 Unions with type tags

It is common practice to treat unions like "dynamic types". But it is common practice to leave a hint in a tag to what type is actually being stored in that union.

```
#include <stdio.h>
#include <stdint.h>
#include <limits.h>
#include <stdlib.h>

// Multitype is either NOTHING, AN_INT, or A_FLOAT
// You should read it and write it based on its type (which)
struct multi_type {
```

```
    enum {
        NOTHING,
        AN_INT,
        A_FLOAT
    } which;
    union {
        int32_t an_int;
        float a_float;
    } value;
};
typedef struct multi_type MultiType;

void print_mt(MultiType mt) {
    if (mt.which == NOTHING) {
        printf("nothing");
    } else if (mt.which == AN_INT) {
        printf("%d", (int) mt.value.an_int);
    } else if (mt.which == A_FLOAT) {
        printf("%e", mt.value.a_float);
    } else {
        abort();
    }
}

void print_mt_array(MultiType *mt_array, size_t length) {
    for (size_t idx = 0; idx < length; idx++) {
        print_mt(mt_array[idx]);
        printf(" ");
    }
    printf("\n");
}
#define EXAMPLES 7
int main() {
    MultiType mt_array[EXAMPLES] = {
        { NOTHING },
        { AN_INT, { .an_int=10 } },
        { A_FLOAT, { .a_float=0.1 } },
        { NOTHING },
        { A_FLOAT, { .a_float=99.9 } },
        { AN_INT, { .an_int=99.9 } },
```

```
        { AN_INT, { .a_float=-99.9 } },
    };
    printf("\n");
    printf("sizeof(mt_array)    == %lu\n",sizeof(mt_array));
    printf("sizeof(mt_array[0]) == %lu\n",sizeof(mt_array[0]));
    printf("sizeof(mt_array[1]) == %lu\n",sizeof(mt_array[1]));
    printf("sizeof(mt_array[2]) == %lu\n",sizeof(mt_array[2]));
    print_mt_array(mt_array, EXAMPLES);
}
```

```
sizeof(mt_array)    == 56
sizeof(mt_array[0]) == 8
sizeof(mt_array[1]) == 8
sizeof(mt_array[2]) == 8
nothing 10 1.000000e-01 nothing 9.990000e+01 99 -1027093299
```

1.7 Malloc! The Heap!

Memory!

Your programs use the following kinds of memory:

- Code: this is for constants and compiled code for the CPU to run
- Data: this is for strings, literals, and other values you predefine in your program.
- Stack: this is where the data for your function locals goes
- Heap: this is where dynamically allocated memory goes. It is the largest pool.

What memory does our program use? (OS and compiler specific)

- Globals? Data.
- Static variables? Data.
- Constants? Code and or Data
- Local variables? stack
- Dynamic allocation? heap

1.7.1 On my computer

Here's what emacs is using

```
root@st-francis:/proc/27627# cat maps
00400000-00641000 r-xp 00000000 09:00 116130283                          /usr/bin/emacs
00841000-00848000 r--p 00241000 09:00 116130283                          /usr/bin/emacs
00848000-01615000 rw-p 00248000 09:00 116130283                          /usr/bin/emacs
03155000-0d208000 rw-p 00000000 00:00 0                                  [heap]
7f16739f4000-7f1673a74000 rw-s 00000000 00:05 935100458                  /SYSV00000000
7f1673a74000-7f1673a79000 r-xp 00000000 09:00 394799                     /usr/lib/x86_6
7f1673a79000-7f1673c78000 ---p 00005000 09:00 394799                     /usr/lib/x86_6
7f1673c78000-7f1673c79000 r--p 00004000 09:00 394799                     /usr/lib/x86_6
...
7f168942a000-7f168942b000 r--p 00027000 09:00 103024636                  /lib/x86_64-li
7f168942b000-7f168942c000 rw-p 00028000 09:00 103024636                  /lib/x86_64-li
7f168942c000-7f168942d000 rw-p 00000000 00:00 0
7fffcf9ad000-7fffcfa6d000 rw-p 00000000 00:00 0                          [stack]
7fffcfb29000-7fffcfb2c000 r--p 00000000 00:00 0                          [vvar]
7fffcfb2c000-7fffcfb2e000 r-xp 00000000 00:00 0                          [vdso]
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0                  [vsyscall]
```

Here's the important lines

```
code        00400000-00641000 r-xp 00000000 09:00 116130283                      /u
data?       00841000-00848000 r--p 00241000 09:00 116130283                      /u
data?       00848000-01615000 rw-p 00248000 09:00 116130283                      /u
heap        03155000-0d208000 rw-p 00000000 00:00 0                              [
stack       7fffcf9ad000-7fffcfa6d000 rw-p 00000000 00:00 0                      [s
```

So stack is limited

```
hindle1@st-francis:~$ ulimit -a
core file size          (blocks, -c) 0
data seg size           (kbytes, -d) unlimited
scheduling priority             (-e) 0
file size               (blocks, -f) unlimited
pending signals                 (-i) 273535
max locked memory       (kbytes, -l) 16384
max memory size         (kbytes, -m) unlimited
open files                      (-n) 1024
pipe size            (512 bytes, -p) 8
```

```
POSIX message queues     (bytes, -q) 819200
real-time priority              (-r) 0
stack size              (kbytes, -s) 8192
cpu time               (seconds, -t) unlimited
max user processes              (-u) 273535
virtual memory          (kbytes, -v) unlimited
file locks                      (-x) unlimited

stack size              (kbytes, -s) 8192
```

8megs of stack.
What if I want a big array?
I can tell bash to give me more, but sometimes you are limited.
How do programs using more than 8mb of memory?
THE HEAP!!
How do I get heap memory?
malloc!
Can I get it any time.
Sure.

1.7.2 Malloc

Just stack allocation

```
#include <assert.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void allocAndGo(const int len) {
        int bigArray[len];
        for(int idx=0; idx < len; idx++) {
            bigArray[idx] = idx;
        }
        printf("%u ints allocated!\n",1+bigArray[len-1]);
        printf("%lu bytes!\n", sizeof(int)*len);
        printf("%p\n", (void*)bigArray);
}
int main() {
    // let's find the max of the stack.
    for (int i = 1; i < 2*900000; i+=256*1024) {
```

```
        allocAndGo(i);
    }
}
```

```
1 ints allocated!
4 bytes!
0x7ffe30690e80
262145 ints allocated!
1048580 bytes!
0x7ffe30590e80
524289 ints allocated!
2097156 bytes!
0x7ffe30490e80
786433 ints allocated!
3145732 bytes!
0x7ffe30390e80
1048577 ints allocated!
4194308 bytes!
0x7ffe30290e80
1310721 ints allocated!
5242884 bytes!
0x7ffe30190e80
1572865 ints allocated!
6291460 bytes!
0x7ffe30090e80
```

```
#include <assert.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int * testAllocArray(int arrLen) {
  size_t size = arrLen * sizeof(int);
  int* array = malloc( size );
  assert(array!=NULL);
  memset((void*)array, 0, size);
  /* for(int idx=0; idx<arrLen; idx++) {
    array[idx] = idx;
  } */
  printf("%p\n", (void*)array);
```

```
  printf("%p\n", (void*)&arrLen);
  return array;
}
int main() {
    for (int i = 1; i < 90000000; i+=5*1024*1024) {
        int * bigArray = testAllocArray( i );
        printf("%u ints allocated!\n",1+bigArray[i-1]);
        printf("%lu bytes!\n", sizeof(int)*i);
        free(bigArray); // remember to free it when done!
    }
}
```

```
0x55a5b13ef260
0x7ffde9443eec
1 ints allocated!
4 bytes!
0x7f6e5ae4f010
0x7ffde9443eec
1 ints allocated!
20971524 bytes!
0x7f6e59a4f010
0x7ffde9443eec
1 ints allocated!
41943044 bytes!
0x7f6e5864f010
0x7ffde9443eec
1 ints allocated!
62914564 bytes!
0x7f6e5724f010
0x7ffde9443eec
1 ints allocated!
83886084 bytes!
0x7f6e55e4f010
0x7ffde9443eec
1 ints allocated!
104857604 bytes!
0x7f6e54a4f010
0x7ffde9443eec
1 ints allocated!
125829124 bytes!
```

```
0x7f6e5364f010
0x7ffde9443eec
1 ints allocated!
146800644 bytes!
0x7f6e5224f010
0x7ffde9443eec
1 ints allocated!
167772164 bytes!
0x7f6e50e4f010
0x7ffde9443eec
1 ints allocated!
188743684 bytes!
0x7f6e4fa4f010
0x7ffde9443eec
1 ints allocated!
209715204 bytes!
0x7f6e4e64f010
0x7ffde9443eec
1 ints allocated!
230686724 bytes!
0x7f6e4d24f010
0x7ffde9443eec
1 ints allocated!
251658244 bytes!
0x7f6e4be4f010
0x7ffde9443eec
1 ints allocated!
272629764 bytes!
0x7f6e4aa4f010
0x7ffde9443eec
1 ints allocated!
293601284 bytes!
0x7f6e4964f010
0x7ffde9443eec
1 ints allocated!
314572804 bytes!
0x7f6e4824f010
0x7ffde9443eec
1 ints allocated!
335544324 bytes!
```

```
0x7f6e46e4f010
0x7ffde9443eec
1 ints allocated!
356515844 bytes!
```

1.7.3 Malloc2

Big allocation!

```
gcc -std=c99 -Wall -pedantic -Werror -o board ./board.c
./board | wc

8192    8192 67117056
```

```
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h> // <-- malloc is in stdlib.h

#define KIBI 1024L
#define MEGA (KIBI*KIBI)
#define SIZE 1024*8

/*
 * malloc: Memory ALLOCate, in number of bytes
 * free: deallocate the memory
 *  takes the pointer returned by malloc
 *
 * Memory still needs to be initialized!
 */

uint8_t * get_board() {
    void * allocated = malloc(sizeof(uint8_t) * SIZE * SIZE);
    if (allocated == NULL) {
        printf("Error: Out of memory!\n");
        abort();
    }
    return allocated;
}

int main() {
    uint8_t (*board)[SIZE] = NULL;
```

```
    size_t total_size = sizeof(uint8_t) * SIZE * SIZE;
    board = (uint8_t (*)[SIZE])get_board();
    // board = malloc(total_size);
    for (size_t row = 0; row < SIZE; row++) {
        for (size_t col = 0; col < SIZE; col++) {
            board[row][col] = rand() % 26 + 'A';
        }
    }

    for (size_t row = 0; row < SIZE; row++) {
        for (size_t col = 0; col < SIZE; col++) {
            printf("%c", (char) board[row][col]);
        }
        printf("\n");
    }
    printf("board is %zu mebibytes!\n", total_size/MEGA);
    free(board);
    int * ptr = (int*)NULL;
    *ptr;
    for (size_t row = 0; row < SIZE; row++) {
        for (size_t col = 0; col < SIZE; col++) {
            printf("%c", (char) board[row][col]);
        }
        printf("\n");
    }
}
```

1.7.4 Calloc

Calloc is like malloc except it will initialize the memory for you! Just to 0 though. Which is good enough.

Calloc looks different

man calloc says

void *calloc(size$_t$ nmemb, size$_t$ size);

It's not void, it's void * so you have to cast.

```
#include <assert.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
```

```
int * testAllocArray(int arrLen) {
  int* array = calloc( sizeof(int), arrLen );
  assert(array!=NULL);
  for(int idx=0; idx<arrLen; idx++) {
    array[idx] = idx;
  }
  return array;
}
int main() {
    for (int i = 1; i < 90000000; i+=5*1024*1024) {
        int * bigArray = testAllocArray( i );
        printf("%u ints allocated!\n",1+bigArray[i-1]);
        printf("%lu bytes!\n", sizeof(int)*i);
        free(bigArray); // remember to free it when done!
    }
}
```

```
1 ints allocated!
4 bytes!
5242881 ints allocated!
20971524 bytes!
10485761 ints allocated!
41943044 bytes!
15728641 ints allocated!
62914564 bytes!
20971521 ints allocated!
83886084 bytes!
26214401 ints allocated!
104857604 bytes!
31457281 ints allocated!
125829124 bytes!
36700161 ints allocated!
146800644 bytes!
41943041 ints allocated!
167772164 bytes!
47185921 ints allocated!
188743684 bytes!
52428801 ints allocated!
209715204 bytes!
```

```
57671681 ints allocated!
230686724 bytes!
62914561 ints allocated!
251658244 bytes!
68157441 ints allocated!
272629764 bytes!
73400321 ints allocated!
293601284 bytes!
78643201 ints allocated!
314572804 bytes!
83886081 ints allocated!
335544324 bytes!
89128961 ints allocated!
356515844 bytes!
```

1.7.5 strdup

strdup duplicates a string into newly malloc'd memory.

Very handy.

Very dangerous.

strlen's your input string malloc's your input string size + 1 strcpy's your input to the malloc'd location

```
#define _POSIX_C_SOURCE 200809L // <-- needed for strdup
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h> // <-- strdup is in string.h

/*
 * strdup = malloc + strcpy
 */

int main() {
    const char * message = "hello, world!";
    char buffer[14];
    strncpy(buffer, message, 14);
    printf("%p %s\n", (void*)message, message);
    printf("%p %s\n", (void*)buffer, buffer);
```

```
    // hi this code is basically strdup
    char * copyMalloc = malloc((strlen(message) + 1) * sizeof(char));
    strcpy(copyMalloc, message);
    // ^^^^ that was basically strdup

    printf("%p %s\n", (void*)copyMalloc, copyMalloc);
    char * copyDup = strdup(message);
    printf("%p %s\n", (void*)copyDup, copyDup);
    // > .c:30:16: error: assignment of read-only location ‘*message’
    // >    message[0] = 'H';
    // message[0] = 'H';
    copyDup[0] = 'J';
    copyMalloc[0] = 'M';
    printf("%s\n", message);
    printf("%s\n", copyMalloc);
    printf("%s\n", copyDup);
    free(copyDup);
    free(copyMalloc);
}
```

```
0x5641ca73ba84 hello, world!
0x7ffe49df38da hello, world!
0x5641cc91d270 hello, world!
0x5641cc91d290 hello, world!
hello, world!
Mello, world!
Jello, world!
```

1.7.6 free

What happens if we don't free?

Our program can get bigger!

```
#include <assert.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int * testAllocArray(int arrLen) {
  int* array = calloc( sizeof(int), arrLen );
```

```
  assert(array!=NULL);
  for(int idx=0; idx<arrLen; idx++) {
    array[idx] = idx;
  }
  return array;
}
int main() {
    for (int i = 1; i < 10000000; i+=1*1024*1024) {
        int * bigArray = testAllocArray( i );
        printf("%u ints allocated!\n",1+bigArray[i-1]);
        printf("%lu bytes!\n", sizeof(int)*i);
        // free(bigArray); // remember to free it when done!
    }
}
```

```
1 ints allocated!
4 bytes!
1048577 ints allocated!
4194308 bytes!
2097153 ints allocated!
8388612 bytes!
3145729 ints allocated!
12582916 bytes!
4194305 ints allocated!
16777220 bytes!
5242881 ints allocated!
20971524 bytes!
6291457 ints allocated!
25165828 bytes!
7340033 ints allocated!
29360132 bytes!
8388609 ints allocated!
33554436 bytes!
9437185 ints allocated!
37748740 bytes!
```

Valgrind is a memory leak detector. It analyzes memory allocations and warns us about mistakes.

Valgrind will show us that we're leaking memory (losing track of it and not freeing it).

```
gcc -std=c99 -Wall -pedantic -Werror -o nofree ./nofree.c
valgrind ./nofree 2>&1
echo now let\'s leak check
valgrind --leak-check=full ./nofree 2>&1

==29507== Memcheck, a memory error detector
==29507== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==29507== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==29507== Command: ./nofree
==29507==
1 ints allocated!
4 bytes!
1048577 ints allocated!
4194308 bytes!
2097153 ints allocated!
8388612 bytes!
3145729 ints allocated!
12582916 bytes!
4194305 ints allocated!
16777220 bytes!
5242881 ints allocated!
20971524 bytes!
6291457 ints allocated!
25165828 bytes!
7340033 ints allocated!
29360132 bytes!
8388609 ints allocated!
33554436 bytes!
9437185 ints allocated!
37748740 bytes!
==29507==
==29507== HEAP SUMMARY:
==29507==     in use at exit: 188,743,720 bytes in 10 blocks
==29507==   total heap usage: 11 allocs, 1 frees, 188,747,816 bytes allocated
==29507==
==29507== LEAK SUMMARY:
==29507==    definitely lost: 100,663,320 bytes in 6 blocks
==29507==    indirectly lost: 0 bytes in 0 blocks
==29507==      possibly lost: 88,080,400 bytes in 4 blocks
==29507==    still reachable: 0 bytes in 0 blocks
```

```
==29507==         suppressed: 0 bytes in 0 blocks
==29507== Rerun with --leak-check=full to see details of leaked memory
==29507==
==29507== For counts of detected and suppressed errors, rerun with: -v
==29507== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
now let's leak check
==29508== Memcheck, a memory error detector
==29508== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==29508== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==29508== Command: ./nofree
==29508==
1 ints allocated!
4 bytes!
1048577 ints allocated!
4194308 bytes!
2097153 ints allocated!
8388612 bytes!
3145729 ints allocated!
12582916 bytes!
4194305 ints allocated!
16777220 bytes!
5242881 ints allocated!
20971524 bytes!
6291457 ints allocated!
25165828 bytes!
7340033 ints allocated!
29360132 bytes!
8388609 ints allocated!
33554436 bytes!
9437185 ints allocated!
37748740 bytes!
==29508==
==29508== HEAP SUMMARY:
==29508==     in use at exit: 188,743,720 bytes in 10 blocks
==29508==   total heap usage: 11 allocs, 1 frees, 188,747,816 bytes allocated
==29508==
==29508== 88,080,400 bytes in 4 blocks are possibly lost in loss record 1 of 2
==29508==    at 0x4C31B25: calloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux
==29508==    by 0x1086F6: testAllocArray (in /home/hindle1/projects/CMPUT201W20/2020-01
==29508==    by 0x10876F: main (in /home/hindle1/projects/CMPUT201W20/2020-01/CMPUT201W
```

```
==29508==
==29508== 100,663,320 bytes in 6 blocks are definitely lost in loss record 2 of 2
==29508==    at 0x4C31B25: calloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux
==29508==    by 0x1086F6: testAllocArray (in /home/hindle1/projects/CMPUT201W20/2020-0
==29508==    by 0x10876F: main (in /home/hindle1/projects/CMPUT201W20/2020-01/CMPUT201
==29508==
==29508== LEAK SUMMARY:
==29508==    definitely lost: 100,663,320 bytes in 6 blocks
==29508==    indirectly lost: 0 bytes in 0 blocks
==29508==      possibly lost: 88,080,400 bytes in 4 blocks
==29508==    still reachable: 0 bytes in 0 blocks
==29508==         suppressed: 0 bytes in 0 blocks
==29508==
==29508== For counts of detected and suppressed errors, rerun with: -v
==29508== ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 0 from 0)
```

1.7.7 realloc!

realloc does a lot of work for you!

It will use the information that malloc uses to see if it can just leave the pointer in place and safely give it more space.

If there's not enough space it will allocate a new region of memory and return that new pointer. It will free the old pointer if that was the case.

There's no guarantee that you pointer stays in the same spot!

- With realloc, you must replace the old pointer with the new one!
- With realloc you must check if there was enough memory to do it!
  - check return value!

```
#include <assert.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

// We're going to malloc 3 regions and keep growing it
// by 5 characters
int main() {
    char * array1 = (char*)malloc(10);
    char * array2 = (char*)malloc(20);
```

```
    char * array3 = (char*)malloc(30);
    for (int i = 30; i < 500; i+=5) {
        char * newArray1 = realloc(array1, i * sizeof(char));
        char * newArray2 = realloc(array2, i * sizeof(char));
        char * newArray3 = realloc(array3, i * sizeof(char));
        assert(newArray1 != NULL || newArray2 != NULL || newArray3 != NULL);
        if ( array1 != newArray1 )  {
            printf("size: %05d array1: old: %p new: %p\n", i, array1, newArray1);
        }
        /*
        if ( array2 != newArray2 )  {
            printf("size: %05d array2: old: %p new: %p\n", i, array2, newArray2);
        }
        if ( array3 != newArray3 )  {
            printf("size: %05d array3: old: %p new: %p\n", i, array3, newArray3);
        }
        */
        // YOU MUST REPLACE THE OLD VALUE, IT IS DANGEROUS!
        array1 = newArray1;
        array2 = newArray2;
        array3 = newArray3;
    }
    free(array1);
    free(array2);
    free(array3);
}
```

```
size: 00030 array1: old: 0x55a6033c2260 new: 0x55a6033c22d0
size: 00045 array1: old: 0x55a6033c22d0 new: 0x55a6033c3340
size: 00060 array1: old: 0x55a6033c3340 new: 0x55a6033c3400
size: 00075 array1: old: 0x55a6033c3400 new: 0x55a6033c34f0
size: 00090 array1: old: 0x55a6033c34f0 new: 0x55a6033c3610
size: 00105 array1: old: 0x55a6033c3610 new: 0x55a6033c3760
size: 00125 array1: old: 0x55a6033c3760 new: 0x55a6033c38e0
size: 00140 array1: old: 0x55a6033c38e0 new: 0x55a6033c3a90
size: 00155 array1: old: 0x55a6033c3a90 new: 0x55a6033c3c70
size: 00170 array1: old: 0x55a6033c3c70 new: 0x55a6033c3e80
size: 00185 array1: old: 0x55a6033c3e80 new: 0x55a6033c40c0
size: 00205 array1: old: 0x55a6033c40c0 new: 0x55a6033c4330
size: 00220 array1: old: 0x55a6033c4330 new: 0x55a6033c45d0
```

```
size: 00235 array1: old: 0x55a6033c45d0 new: 0x55a6033c48a0
size: 00250 array1: old: 0x55a6033c48a0 new: 0x55a6033c4ba0
size: 00265 array1: old: 0x55a6033c4ba0 new: 0x55a6033c4ed0
size: 00285 array1: old: 0x55a6033c4ed0 new: 0x55a6033c5230
size: 00300 array1: old: 0x55a6033c5230 new: 0x55a6033c55c0
size: 00315 array1: old: 0x55a6033c55c0 new: 0x55a6033c5980
size: 00330 array1: old: 0x55a6033c5980 new: 0x55a6033c5d70
size: 00345 array1: old: 0x55a6033c5d70 new: 0x55a6033c6190
size: 00365 array1: old: 0x55a6033c6190 new: 0x55a6033c65e0
size: 00380 array1: old: 0x55a6033c65e0 new: 0x55a6033c6a60
size: 00395 array1: old: 0x55a6033c6a60 new: 0x55a6033c6f10
size: 00410 array1: old: 0x55a6033c6f10 new: 0x55a6033c73f0
size: 00425 array1: old: 0x55a6033c73f0 new: 0x55a6033c7900
size: 00445 array1: old: 0x55a6033c7900 new: 0x55a6033c7e40
size: 00460 array1: old: 0x55a6033c7e40 new: 0x55a6033c83b0
size: 00475 array1: old: 0x55a6033c83b0 new: 0x55a6033c8950
size: 00490 array1: old: 0x55a6033c8950 new: 0x55a6033c8f20
```

1.7.8 Malloc and structs

Mallocs are often used with arrays of structs. You need to get the sizeof the struct.

```
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

enum card_face {
    ACE = 1,
    FACE2,
    FACE3,
    FACE4,
    FACE5,
    FACE6,
    FACE7,
    FACE8,
    FACE9,
    FACE10,
    JACK,
```

```
    QUEEN,
    KING,
};

typedef enum card_face CardFace;

#define NFACES 13
#define NFACEOFF 1

enum card_suit {
    CLUBS,
    HEARTS,
    DIAMONDS,
    SPADES
};

typedef enum card_suit CardSuit;

#define NSUIT 4

struct playing_card {
    CardFace face;
    CardSuit suit;
};

typedef struct playing_card PlayingCard;

#define HANDSIZE 5

bool isFlush(PlayingCard hand[HANDSIZE]) {
    CardSuit suit = hand[0].suit;
    for (int i = 1;  i < HANDSIZE; i++ ) {
        if (suit != hand[i].suit) {
            return false;
        }
    }
    return true;
}

PlayingCard randomCard() {
```

```
    PlayingCard card = {ACE, CLUBS};
    card.face = NFACEOFF + ( rand() % NFACES );
    card.suit =  rand() % NSUIT;
    return card;
}
int main() {
    srand(time(NULL));
    const int N = 1000000;
    PlayingCard * bigHand = malloc(sizeof(PlayingCard)*N);
    for (int i = 0; i < N; i++) {
        bigHand[i] = randomCard();
    }
    int flushes = 0;
    for (int i = 0; i < N - HANDSIZE; i+=HANDSIZE) {
        if (isFlush(bigHand + i)) {
            if (flushes < 10) { // reduce printing
                printf("Flush found at card %d\n", i);
                printf("Suit %d\n", bigHand[i].suit);
            }
            flushes++;
        }
    }
    printf("We found %d flushes out of %d hands: %f\n", flushes, N/HANDSIZE, flushes/(
}
```

```
Flush found at card 10
Suit 2
Flush found at card 1720
Suit 1
Flush found at card 1940
Suit 2
Flush found at card 3200
Suit 3
Flush found at card 3380
Suit 0
Flush found at card 3950
Suit 0
Flush found at card 5170
```

```
Suit 1
Flush found at card 5510
Suit 1
Flush found at card 6480
Suit 1
Flush found at card 8080
Suit 1
We found 773 flushes out of 200000 hands: 0.003865
```

1.7.9 Malloc 2D arrays

How does Malloc work with 2D arrays? Well 2D arrays are tightly packed so it is pretty easy to determine their size in memory.

```
// READ man 3 malloc

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

// Option 3:
// In functions... just a 2-D array as normal...
// ...we could use either int a[n][n] OR
// int a[][n], but NOT a[][]!
void print_2d(size_t n, int a[][n]) {
    for (size_t i = 0; i < n; i++) {
        for (size_t j = 0; j < n; j++) {
            printf("%2d ", a[i][j]);
        }
        printf("\n");
    }
}

int main(int argc, char **argv) {
    size_t n = 10;
    int * array = (int *) // cast result of malloc to "ptr to an int"
        malloc(n * n * sizeof(int));
    int (*array2d)[n] = (int (*)[n]) array;
    int k = 0;
    for (size_t i = 0; i < n; i++) {
```

```
        for (size_t j = 0; j < n; j++) {
            // Option 1:
            // Use a 1-D array and arithmetic
            array[i * n + j] = k++;
        }
    }
    printf("Printing Option2\n\n");
    for (size_t i = 0; i < n; i++) {
        for (size_t j = 0; j < n; j++) {
            // Option 2:
            // Use 2-D array
            printf("%2d ", array2d[i][j]);
        }
        printf("\n");
    }
    printf("\nPrinting Option3\n\n");
    print_2d(n, (int(*)[n]) array);
    free(array); // deallocates or "frees" the memory we were using for array
    // Now ALL pointers to or into the array are invalid!
}
```

```
Printing Option2

 0  1  2  3  4  5  6  7  8  9
10 11 12 13 14 15 16 17 18 19
20 21 22 23 24 25 26 27 28 29
30 31 32 33 34 35 36 37 38 39
40 41 42 43 44 45 46 47 48 49
50 51 52 53 54 55 56 57 58 59
60 61 62 63 64 65 66 67 68 69
70 71 72 73 74 75 76 77 78 79
80 81 82 83 84 85 86 87 88 89
90 91 92 93 94 95 96 97 98 99

Printing Option3

 0  1  2  3  4  5  6  7  8  9
10 11 12 13 14 15 16 17 18 19
20 21 22 23 24 25 26 27 28 29
30 31 32 33 34 35 36 37 38 39
```

```
40 41 42 43 44 45 46 47 48 49
50 51 52 53 54 55 56 57 58 59
60 61 62 63 64 65 66 67 68 69
70 71 72 73 74 75 76 77 78 79
80 81 82 83 84 85 86 87 88 89
90 91 92 93 94 95 96 97 98 99
```

1.7.10 Malloc Array of Array versus 2D

So instead of allocating a big block and carving a 2D array out of it we could just allocate each row and make an array of arrays.

Try playing with the order of allocation of rows. Does it affect the result?

```
#include <stdio.h>
#include <stdlib.h>

// This example compares using malloc to get space for a 2-D array vs using malloc to m

int * alloc2d(size_t n) {
    // we can just do 1 malloc()
    return (int *) malloc(n * n * sizeof(int));
}

int ** alloc_aoa(size_t n) {
    // we have to do 1 + n malloc()s
    int ** p = malloc(n * sizeof(int *));
    // we don't need to do them in order...
    for (size_t i = 0; i < n; i++) {
        p[i] = malloc(n * sizeof(int));
    }
    return p;
}

void free2d(int * p) {
    // we can just do 1 free()
    free(p);
}

void free_aoa(size_t n, int ** p) {
    // we have to do n + 1 free()s
```

```
    for (size_t i = 0; i < n; i++) {
        free(p[i]);
    }
    free(p);
}

int get2d(size_t n, int * p, size_t i, size_t j) {
    return p[i * n + j];
}

int get_aoa(int **p, size_t i, size_t j) {
    return p[i][j];
}

int set2d(size_t n, int * p, size_t i, size_t j, int v) {
    return p[i * n + j] = v;
}

int set_aoa(int **p, size_t i, size_t j, int v) {
    return p[i][j] = v;
}

int main(int argc, char **argv) {
    srand(1);
    printf("I'm going to make space for a big, square table in memory.\n");
    printf("How many rows and columns would you like to make space for? ");
    size_t n;
    // int r = scanf("%zu", &n);
    n = 30;
    if (n != 1) {
        printf("Sorry, I couldn't understand that :(\n");
    }
    // allocate them
    int *p2d = alloc2d(n);
    int **aoa = alloc_aoa(n);
    // initialize them
    for (size_t i = 0; i < n; i++) {
        for (size_t j = 0; j < n; j++) {
            set2d(n, p2d, i, j, rand() % 10);
            set_aoa(aoa, i, j, rand() % 10);
```

```
        }
    }
    // print them out
    printf("2d:\n");
    for (size_t i = 0; i < n; i++) {
        for (size_t j = 0; j < n; j++) {
            int x = get2d(n, p2d, i, j);
            printf("%d ", x);
        }
        printf("\n");
    }
    printf("aoa:\n");
    for (size_t i = 0; i < n; i++) {
        for (size_t j = 0; j < n; j++) {
            int x = get_aoa(aoa, i, j);
            printf("%d ", x);
        }
        printf("\n");
    }
    // free them
    free2d(p2d);
    free_aoa(n, aoa);
}
```

```
I'm going to make space for a big, square table in memory.
How many rows and columns would you like to make space for? Sorry, I couldn't understa
2d:
3 7 3 6 9 2 0 3 0 2 1 7 2 2 7 9 2 9 3 1 9 1 4 8 5 3 1 6 2 6
5 4 6 6 3 4 2 4 4 3 7 6 8 3 4 2 6 9 6 4 5 4 7 7 7 2 1 6 5 4
0 1 7 1 9 7 7 6 6 9 8 2 3 0 8 0 6 8 6 1 9 4 1 3 4 4 7 3 7 9
2 7 5 4 8 9 5 8 3 8 6 3 3 6 4 8 9 7 4 0 0 2 4 5 4 9 2 7 5 8
2 9 6 0 1 5 1 8 0 4 2 8 2 4 2 0 2 9 8 3 1 3 0 9 9 9 3 0 6 4
0 6 6 5 9 7 8 9 6 2 6 3 1 9 1 9 0 5 7 4 0 2 6 0 2 2 5 2 0 8
8 4 9 9 2 4 9 3 0 0 9 3 1 4 1 6 4 2 4 2 8 2 8 6 3 3 3 0 7 8
0 8 9 3 3 3 6 2 5 7 6 4 0 8 0 6 4 9 9 8 0 7 9 5 9 5 4 9 5 3
7 8 9 7 2 3 9 2 1 6 1 0 3 1 0 6 7 0 4 4 5 2 0 6 6 8 6 7 1 1
7 2 4 2 2 0 9 5 0 7 8 0 6 6 9 5 7 5 3 3 9 7 7 1 0 8 5 4 7 3
0 7 9 2 3 1 2 2 7 1 4 7 1 7 4 8 1 6 1 6 8 8 0 2 7 6 6 7 7 9
7 6 8 3 4 5 1 5 9 3 5 2 7 3 6 6 3 4 9 2 8 0 4 6 7 3 3 5 0 7
3 0 0 1 3 9 4 5 8 5 5 9 7 3 6 5 6 0 1 2 9 0 2 4 3 8 3 0 3 9
```

7 2 2 4 8 0 9 2 1 3 2 4 1 5 1 9 1 3 7 8 7 4 4 1 8 2 9 6 6 9
0 9 1 8 6 7 7 2 1 0 0 0 3 4 1 0 2 7 6 4 2 7 4 6 7 5 2 3 4 9
2 1 3 2 5 5 0 4 6 2 8 5 6 8 7 2 0 8 5 7 8 3 7 7 9 1 0 9 8 3
0 9 1 7 7 2 1 8 4 6 6 4 8 8 5 4 0 7 2 2 3 9 1 5 4 2 1 2 2 9
4 5 1 0 1 7 9 1 7 0 0 5 9 1 1 0 8 4 2 4 9 2 9 0 4 9 5 6 3 9
2 3 9 1 4 8 7 3 9 5 8 0 3 1 7 5 1 3 0 5 2 9 9 9 1 3 3 4 1 6
7 2 2 1 4 8 3 7 3 2 3 6 1 6 0 5 5 9 8 2 9 1 0 6 9 8 8 3 0 5
3 8 1 9 0 5 4 4 9 9 3 3 7 4 9 9 2 6 9 6 1 3 2 3 9 4 4 9 8 2
5 3 4 5 7 9 7 7 9 5 4 7 3 2 2 3 1 8 0 2 9 9 3 8 6 7 7 1 0 4
3 3 7 1 9 6 9 5 1 9 1 2 0 3 1 7 8 0 4 3 9 4 5 2 7 8 9 3 8 4
6 8 5 1 6 8 6 5 6 1 3 5 6 4 6 7 3 9 0 2 9 3 5 7 7 6 4 3 2 6
9 5 3 4 1 1 9 5 2 9 7 4 1 1 8 4 3 3 7 3 8 0 8 8 3 5 5 2 8 2
3 7 7 6 2 7 3 2 5 7 9 1 4 5 8 3 5 1 5 0 8 9 9 6 5 5 0 2 9 2
6 5 8 7 6 2 9 0 7 5 4 0 8 4 4 8 2 6 2 7 4 6 4 4 5 6 3 7 2 0
9 1 4 5 2 0 3 1 5 4 0 3 9 4 3 2 5 8 1 1 8 3 9 5 4 6 2 0 3 7
3 1 4 1 6 3 7 0 4 3 7 9 3 2 9 5 0 3 9 5 3 2 7 7 0 6 5 8 9 7
0 1 3 7 2 1 3 8 8 8 8 9 3 4 7 3 6 2 2 5 4 4 1 3 8 3 9 4 1 0

aoa:

6 5 5 2 1 7 9 6 6 6 8 9 0 3 5 2 8 7 6 2 3 9 7 4 0 6 0 3 0 1
5 7 5 9 7 5 5 7 4 0 8 8 4 1 9 0 8 2 6 9 0 8 1 2 2 6 0 1 9 9
9 7 1 5 7 6 3 5 3 4 1 9 9 8 5 9 3 5 1 5 8 8 0 0 4 4 6 1 5 6
1 8 7 1 5 7 3 8 1 9 4 3 8 0 8 8 7 6 3 3 9 5 0 9 6 2 4 7 4 1
8 3 8 2 0 1 0 5 6 6 5 6 8 7 4 6 9 0 1 1 0 4 3 1 6 3 8 5 6 0
4 2 7 6 8 2 2 9 0 7 1 2 5 9 4 1 7 8 0 8 4 9 1 4 2 0 5 9 2 3
0 0 1 6 5 4 9 6 5 2 4 5 7 3 4 9 2 6 1 8 9 8 8 8 8 3 8 4 6 9
6 7 0 3 7 2 5 6 8 9 0 1 4 7 8 2 7 3 2 3 1 8 1 4 2 7 9 4 9 5
0 1 9 8 5 4 0 0 9 2 2 7 1 9 5 7 4 6 7 8 8 6 6 4 2 9 0 0 0 3
7 6 5 0 9 9 4 1 3 8 6 4 7 0 7 9 8 3 8 7 3 8 4 9 9 8 8 3 1 8
9 9 3 4 7 2 0 1 5 7 1 1 1 0 0 5 6 2 9 4 0 1 2 9 5 4 3 9 4 1
0 0 5 9 1 4 5 4 8 8 2 2 0 4 3 3 4 3 7 5 9 2 7 5 1 3 8 1 8 6
5 8 4 1 5 3 1 0 3 6 9 0 6 7 1 0 5 8 2 6 1 4 7 0 2 0 7 0 4 2
4 5 4 3 6 8 2 3 8 4 2 5 7 7 6 8 3 3 9 6 0 8 8 6 5 1 9 0 4 9
8 3 4 9 7 3 1 2 5 9 4 1 7 1 3 3 1 5 5 2 1 2 1 5 8 9 7 6 7 7
2 6 0 1 6 0 3 6 0 5 9 0 0 3 8 1 5 5 0 3 2 0 7 6 1 9 8 8 0 7
6 2 7 9 6 7 5 8 5 5 8 8 3 7 2 5 5 3 7 1 4 4 9 7 1 2 6 0 2 7
3 6 4 3 2 7 8 0 6 1 2 1 7 3 2 6 7 9 4 5 1 8 6 6 0 4 4 6 9 5
1 0 9 3 5 5 3 8 5 3 6 3 6 8 0 1 0 0 4 4 4 9 4 8 6 9 3 6 5 1
2 9 8 2 7 6 7 2 7 5 7 8 3 4 3 8 0 9 0 4 0 2 0 3 0 3 7 1 0 0
1 0 7 1 3 9 8 6 2 0 0 3 9 9 1 4 0 5 5 1 4 7 7 3 2 4 9 3 3 9
4 9 9 5 3 0 2 2 0 0 1 9 6 1 5 9 8 7 5 7 1 6 6 4 6 2 4 0 6 4

```
7 4 2 7 5 8 5 2 5 9 6 1 5 2 9 6 2 6 3 6 0 8 1 9 3 0 2 1 7 1
3 5 0 2 4 5 2 2 9 3 1 2 9 4 0 4 7 0 2 6 0 5 8 1 0 0 1 0 9 0
3 4 6 3 9 0 4 6 5 1 7 1 9 3 7 9 1 8 9 8 4 0 6 2 8 0 9 6 5 8
6 8 2 6 9 0 7 3 1 8 4 6 3 4 7 3 0 4 7 7 9 3 4 4 5 6 6 6 9 9
5 3 6 3 0 6 3 8 6 2 0 6 5 9 6 3 3 2 4 0 9 5 6 2 1 1 7 1 1 8
0 3 8 8 2 6 6 0 7 2 0 3 0 3 4 4 3 1 3 5 1 3 7 4 9 7 1 1 7 6
9 0 1 8 4 4 7 7 5 0 2 9 0 7 9 2 8 5 6 6 0 0 4 3 1 7 7 8 0 8
3 0 6 3 2 5 3 2 5 0 6 3 7 3 1 9 4 0 9 7 6 9 2 1 1 8 2 5 0 1
```

1.7.11 Malloc array of arrays structs?

```
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

enum card_face {
    ACE = 1,
    FACE2,
    FACE3,
    FACE4,
    FACE5,
    FACE6,
    FACE7,
    FACE8,
    FACE9,
    FACE10,
    JACK,
    QUEEN,
    KING,
};

typedef enum card_face CardFace;

#define NFACES 13
#define NFACEOFF 1

enum card_suit {
    CLUBS,
    HEARTS,
```

```
    DIAMONDS,
    SPADES
};

typedef enum card_suit CardSuit;

#define NSUIT 4

struct playing_card {
    CardFace face;
    CardSuit suit;
};

typedef struct playing_card PlayingCard;

#define HANDSIZE 5

bool isFlush(PlayingCard hand[HANDSIZE]) {
    CardSuit suit = hand[0].suit;
    for (int i = 1;  i < HANDSIZE; i++ ) {
        if (suit != hand[i].suit) {
            return false;
        }
    }
    return true;
}

PlayingCard randomCard() {
    PlayingCard card = {ACE, CLUBS};
    card.face = NFACEOFF + ( rand() % NFACES );
    card.suit =  rand() % NSUIT;
    return card;
}
int main() {
    srand(time(NULL));
    const int HANDS = 1000000;
    PlayingCard * hands = malloc(sizeof(PlayingCard)*HANDS*HANDSIZE);
    for (int i = 0; i < HANDS*HANDSIZE; i++) {
        hands[i] = randomCard();
    }
```

```
    int flushes = 0;
    for (int i = 0; i < HANDS; i++) {
        if (isFlush(hands + i*HANDSIZE)) {
            if (flushes < 10) { // reduce printing
                printf("Flush found at card %d\n", i);
                printf("Suit %d\n", hands[i].suit);
            }
            flushes++;
        }
    }
    printf("We found %d flushes out of %d hands: %f\n", flushes, HANDS, flushes/(float)
}
```

```
Flush found at card 404
Suit 2
Flush found at card 432
Suit 3
Flush found at card 657
Suit 1
Flush found at card 678
Suit 1
Flush found at card 1611
Suit 0
Flush found at card 1671
Suit 2
Flush found at card 1728
Suit 3
Flush found at card 1940
Suit 0
Flush found at card 1982
Suit 1
Flush found at card 2123
Suit 0
We found 3900 flushes out of 1000000 hands: 0.003900
```

That's kind of gross, let's model our hands as arrays of 5 cards instead.

```
#include <stdbool.h>
```

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

enum card_face {
    ACE = 1,
    FACE2,
    FACE3,
    FACE4,
    FACE5,
    FACE6,
    FACE7,
    FACE8,
    FACE9,
    FACE10,
    JACK,
    QUEEN,
    KING,
};

typedef enum card_face CardFace;

#define NFACES 13
#define NFACEOFF 1

enum card_suit {
    CLUBS,
    HEARTS,
    DIAMONDS,
    SPADES
};

typedef enum card_suit CardSuit;

#define NSUIT 4

struct playing_card {
    CardFace face;
    CardSuit suit;
};
```

```
typedef struct playing_card PlayingCard;

#define HANDSIZE 5

bool isFlush(PlayingCard hand[HANDSIZE]) {
    CardSuit suit = hand[0].suit;
    for (int i = 1;  i < HANDSIZE; i++ ) {
        if (suit != hand[i].suit) {
            return false;
        }
    }
    return true;
}

PlayingCard randomCard() {
    PlayingCard card = {ACE, CLUBS};
    card.face = NFACEOFF + ( rand() % NFACES );
    card.suit =  rand() % NSUIT;
    return card;
}
int main() {
    srand(time(NULL));
    const int HANDS = 1000000;
    // Pointer to arrays
    PlayingCard (*hands)[5] = malloc(sizeof(PlayingCard[5])*HANDS);
    for (int i = 0; i < HANDS; i++) {
        for (int j = 0; j < HANDSIZE; j++) {
            hands[i][j] = randomCard();
        }
    }
    int flushes = 0;
    for (int i = 0; i < HANDS; i++) {
        if (isFlush(hands[i])) {
            if (flushes < 10) { // reduce printing
                printf("Flush found at card %d\n", i);
                printf("Suit %d\n", hands[i][0].suit);
            }
            flushes++;
        }
```

```
    }
    printf("We found %d flushes out of %d hands: %f\n", flushes, HANDS, flushes/(float)
}
```

```
Flush found at card 463
Suit 1
Flush found at card 1192
Suit 3
Flush found at card 1282
Suit 1
Flush found at card 1900
Suit 0
Flush found at card 2093
Suit 1
Flush found at card 2234
Suit 3
Flush found at card 2263
Suit 2
Flush found at card 2291
Suit 2
Flush found at card 2503
Suit 2
Flush found at card 2551
Suit 2
We found 3906 flushes out of 1000000 hands: 0.003906
```