CMPUT201W20B2 Week 6

Abram Hindle

March 10, 2020

Contents

1 Week6

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 Structs!

1.5.1 A pointer is not a l value

p = string p++ tmp = p p = (char*)p + 1*sizeof(void)

1.5.2 Struct Intro

How do you store a record?

In python you typically use a dictionary and store key value pairs within a dictionary.

```
my_pet = {"name":"Dan", "type":"dog", "age":6 }
```

In C structures allow you to group relevant information together such that you can access fields (properties).

```
#define NAME_LEN 17
#define TYPE_LEN 9
struct {
    char name[NAME_LEN];
    char type[TYPE_LEN];
    int  age;
} my_pet;
```

In python you access the dictionary or object with:

```
my_pet = {"name":"Dan", "type":"dog", "age":6 }
my_pet["name"]
# or if it was an object
my_pet = Pet("Dan","dog",6)
my_pet.name
```

(&my_{pet})->name $my_{pet.name}$

In C with structs you use the dot operator to access fields or members of the structs

names = { "Darren", "Dan" } ages = { 3 , 6 } type = {"cat", "dog" } 0 1

#define DARREN 0 names[DARREN] ages[DARREN] type[DARREN]

darren.name darren.age darren.type

```
#include <stdio.h>
#define NAME_LEN 17
#define TYPE_LEN 9

int main() {
    struct {
        int x;
        int y;
    } foo;
    struct { // declare a struct type
        char name[NAME_LEN];
        char type[TYPE_LEN];
        double doub;
        int  age;
   } my_pet  = { "Dan", "dog", 5.0, 6 }, // name and initialize 1 instance of the struc
     my_pet2 = { "Dan", "dog", 5.0, 6 }; // name and initialize 1 instance of the struc
   printf("%s, %s, %d\n", my_pet.name, my_pet.type, my_pet.age);
```

```
   printf("sizeof(my_pet)=%lu\n",sizeof(my_pet));
}
```

1.5.3 Structs in memory

Structs are very compact. They are all their field datatypes stacked together.

It will be mixed datatypes in memory:

name	offset	value
name	0	"Dan\0"
type	17	"dog\0"
age	26	6


```
#include <stdio.h>
#define NAME_LEN 17
#define TYPE_LEN 9

int main() {
    struct { // declare a struct type
        char name[NAME_LEN];
        char type[TYPE_LEN];
        int  age;
   } my_pet = { "Dan", "dog", 6 }; // name and initialize 1 instance of the struct
   printf("%s, %s, %d\n", my_pet.name, my_pet.type, my_pet.age);
   printf("struct      location:\t %p\n", (void*)&my_pet);
   printf("my_pet.name location:\t %p\n", (void*)&my_pet.name);
   printf("my_pet.type location:\t %p\n", (void*)&my_pet.type);
   printf("my_pet.age  location:\t %p\n", (void*)&my_pet.age);
   printf("my_pet.name size:\t %lu\n", sizeof(my_pet.name));
   printf("my_pet.type size:\t %lu\n", sizeof(my_pet.type));
   printf("my_pet.age  size:\t %lu\n", sizeof(my_pet.age));
   printf("sizeof(my_pet)=%lu\n",sizeof(my_pet));

}
```

```
Dan, dog, 6
struct      location:  0x7ffcbdf3e0b0
my_pet.name location:  0x7ffcbdf3e0b0
```

```
my_pet.type location:  0x7ffcbdf3e0c1
my_pet.age  location:  0x7ffcbdf3e0cc
my_pet.name size:  17
my_pet.type size:  9
my_pet.age  size:  4
sizeof(my_pet)=32
```

1.5.4 Initializing

```
#include <stdio.h>
#define NAME_LEN 17
#define TYPE_LEN 9

int main() {
    struct { // declare a struct type
        char name[NAME_LEN];
        char type[TYPE_LEN];
        int  age;
   } my_pet1 = { "Dan", "dog", 6 }, // name and initialize 1 instance of the struct
     my_pet2 = { .name = "Darren", .type = "cat", .age = 3 }; // designated initialize
   printf("%s and %s get along just fine.\n", my_pet1.name, my_pet2.name);
}
```

```
Dan and Darren get along just fine.
```

1.5.5 Structure Types

You can predeclare structure "tags" ahead of time so you can reuse your type.

```
#include <stdio.h>
#define NAME_LEN 17
#define TYPE_LEN 9

struct my_pet { // declare a struct type
    char name[NAME_LEN];
    char type[TYPE_LEN];
    int  age;
}; // REMEMBER THE SEMICOLON

int main() {
```

```
    struct my_pet my_pet1 = { "Dan", "dog", 6 }; // name and initialize 1 instance of t
    struct my_pet my_pet2 = { .name = "Darren", .type = "cat", .age = 3 }; // designate
    printf("%s and %s get along just fine.\n", my_pet1.name, my_pet2.name);
}
```

```
Dan and Darren get along just fine.
```

1. Typedef instead of struct tag

   You can also typedef it away but it causes issues later.

   ```
   #include <stdio.h>
   #define NAME_LEN 17
   #define TYPE_LEN 9

   typedef struct { // declare a struct type
       char name[NAME_LEN];
       char type[TYPE_LEN];
       int  age;
   } MyPet; // REMEMBER THE SEMICOLON

   int main() {
       MyPet my_pet1 = { "Dan", "dog", 6 }; // name and initialize 1 instance of the
       MyPet my_pet2 = { .name = "Darren", .type = "cat", .age = 3 }; // designated i
       printf("%s and %s get along just fine.\n", my_pet1.name, my_pet2.name);
   }
   ```

   ```
   Dan and Darren get along just fine.
   ```

2. Or combine both typedef and struct tags

   ```
   #include <stdio.h>
   #define NAME_LEN 17
   #define TYPE_LEN 9

   // First declare the struct tag
   // struct my_pet
   struct my_pet { // declare a struct type
       char name[NAME_LEN];
       char type[TYPE_LEN];
   ```

```
    int  age;
    struct my_pet * ptr;
}; // REMEMBER THE SEMICOLON

// Then typedef it

typedef struct my_pet MyPet;

int main() {
    MyPet my_pet1 = { "Dan", "dog", 6 }; // name and initialize 1 instance of the
    MyPet my_pet2 = { .name = "Darren", .type = "cat", .age = 3 }; // designated i
    printf("%s and %s get along just fine.\n", my_pet1.name, my_pet2.name);
}
```

```
Dan and Darren get along just fine.
```

3. Hazel's example of typedef and style

```
#include <stdio.h>

/* A common thing to do is to typedef a struct
 * so that you don't have to type struct whatever
 * so often.
 */

struct coordinate {
    float x;
    float y;
};

// We use a capital first letter to indicate a type
// This is a newer style.
typedef struct coordinate Coordinate;
// Or we could use "_t" at the end.
// This is an older style. Remember uint64_t?
typedef struct coordinate coordinate_t;

Coordinate move_left(Coordinate position) {
    position.x -= 1.0;
```

```
    return position;
}

int main() {
    Coordinate position = { 0, 0 };
    printf("position=(%g,%g)\n",
           position.x,
           position.y
    );
    Coordinate new_position = move_left(position);
    printf("position=(%g,%g)\n",
           position.x,
           position.y
    );
    printf("new_position=(%g,%g)\n",
           new_position.x,
           new_position.y
    );
    position = move_left(move_left(position));
    printf("position=(%g,%g)\n",
           position.x,
           position.y
    );
}
```

```
position=(0,0)
position=(0,0)
new_position=(-1,0)
position=(-2,0)
```

4. Pass by Value Gotcha

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

/* The important thing to notice here is that
 * structs are pass-by-value. Just like a single float,
 * when we pass a struct to a function it gets a COPY
 * of the original struct!
```

```
 * We can also assign structs and we get a COPY.
 * We can also return structs and we get a COPY.
 */

struct coordinate { // leaving behind for size comparison
    float x;
    float y;
};

struct named_coordinate {
    float x;
    float y;
    char * name; // WARNING this is a pointer!
};

struct named_coordinate move_left(struct named_coordinate position) {
    printf("I am moving position.name: %s [%p] LEFT\n",
           position.name,
           (void*)position.name);
    position.x -= 1.0;
    return position;
}

int main() {
    char * my_string_literal = "YoloStringLiteral!-X-X-X-X-X-X";
    printf("my_string_literal pointer is at %p\n", (void*)(&my_string_literal));
    struct named_coordinate position = { 0, 0, my_string_literal };
    printf("position=(%g,%g)\n",
           position.x,
           position.y
    );
    struct named_coordinate new_position = move_left(position);
    printf("position=(%g,%g)\n",
           position.x,
           position.y
    );
    printf("new_position=(%g,%g)\n",
           new_position.x,
           new_position.y
    );
```

```
    position = move_left(move_left(position));
    printf("position=(%g,%g)\n",
           position.x,
           position.y
    );
    // So we now have a string in our struct? How does it change the struct?
    printf("Size of named_coordinate: %lu\n",sizeof(new_position));
    printf("Size of coordinate: %lu\n",sizeof(struct coordinate));
    printf("Size of my_string_literal: %lu\n",sizeof(my_string_literal));
    printf("strlen of my_string_literal: %lu\n",strlen(my_string_literal));
    // COPY BY VALUE MEANS POINTERS ARE COPIED, but not their contents.
    printf("&position.name     %p\n",(void*)&position.name); // they have differen
    printf("&new_position.name %p\n", (void*)&new_position.name);  // they have di
    printf("position.name      %p\n", (void*)position.name); // but they point to
    printf("new_position.name  %p\n", (void*)new_position.name); // but they point
    printf("position.name      %s\n", position.name); // but they point to the sam
    printf("new_position.name  %s\n", new_position.name); // but they point to the

}
```

```
my_string_literal pointer is at 0x7ffdfbeeaa68
position=(0,0)
I am moving position.name: YoloStringLiteral!-X-X-X-X-X-X [0x55afc2588a98] LEFT
position=(0,0)
new_position=(-1,0)
I am moving position.name: YoloStringLiteral!-X-X-X-X-X-X [0x55afc2588a98] LEFT
I am moving position.name: YoloStringLiteral!-X-X-X-X-X-X [0x55afc2588a98] LEFT
position=(-2,0)
Size of named_coordinate: 16
Size of coordinate: 8
Size of my_string_literal: 8
strlen of my_string_literal: 30
&position.name     0x7ffdfbeeaa78
&new_position.name 0x7ffdfbeeaa88
position.name      0x55afc2588a98
new_position.name  0x55afc2588a98
position.name      YoloStringLiteral!-X-X-X-X-X-X
new_position.name  YoloStringLiteral!-X-X-X-X-X-X
```

1.5.6 Pointers and Structs

```
#include <stdio.h>

/* Using pointers to structs is very common.
 *
 * Using typedef to define a type that is a pointer
 * to a particular kind of struct is also very common
 * to avoid having to write the pointer everywhere.
 *
 * This allows us to make a sort of
 * object-like variable.
 */

struct coordinate {
    float x;
    float y;
};

typedef struct coordinate *Coordinate;
typedef struct coordinate *coordinate_t;

// When we have a pointer to a struct, we use
// "->" instead of "." to talk about a field.

void move_left(Coordinate position) {
    position->x -= 1.0;
}

// "ptr->field" is just shorthand for "(*ptr).field"

void move_up(Coordinate position) {
    position->y -= 1.0;
}

int main() {
    struct coordinate position = { 0, 0 };
    printf("position=(%g,%g)\n",
           position.x,
           position.y
```

```
    );
    move_left(&position);
    move_up(&position);
    printf("position=(%g,%g)\n",
           position.x,
           position.y
    );
}

position=(0,0)
position=(-1,-1)
```

1.5.7 Elaborate Matrix Example

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

/* But we don't just want to avoid duplicate function
 * parameters, we want to avoid duplicate code too!
 * Noticing that our bounds checking code appears
 * twice, let's refactor...
 */

struct matrix {
    int *elements;
    size_t rows;
    size_t cols;
};

typedef struct matrix Matrix;

struct matrix_element {
    Matrix matrix;
    size_t row;
    size_t col;
};

typedef struct matrix_element MatrixElement;

/* We can add our own bounds-checking to C!
```

```
 */

void bounds_check(MatrixElement elt) {
    if (elt.row >= elt.matrix.rows) {
        printf("Error: row index out of bounds!\n");
        abort();
    }
    if (elt.col >= elt.matrix.cols) {
        printf("Error: col index out of bounds!\n");
        abort();
    }
}

int get_element(MatrixElement elt) {
    bounds_check(elt);
    return elt.matrix.elements[
        elt.row * elt.matrix.cols + elt.col
    ];
}

void set_element(
    MatrixElement elt,
    int value
) {
    bounds_check(elt);
    elt.matrix.elements[
        elt.row * elt.matrix.cols + elt.col
    ] = value;
}

void init_matrix(Matrix matrix) {
    // Note we don't have to keep reallocating memory because
    // structs are COPIED
    MatrixElement elt = {matrix, 0, 0};
    for (elt.row = 0; elt.row < matrix.rows; elt.row++) {
        for (elt.col = 0; elt.col < matrix.cols; elt.col++) {
            set_element(elt, 0);
        }
    }
}
```

```
void print_matrix(Matrix matrix) {
    MatrixElement elt = {matrix, 0, 0};
    for (elt.row = 0; elt.row < matrix.rows; elt.row++) {
        for (elt.col = 0; elt.col < matrix.cols; elt.col++) {
            int value = get_element(elt);
            printf("%d ", value);
        }
        printf("\n");
    }
}

int main() {
    size_t rows = 3;
    size_t cols = 3;
    // we will use our init_matrix function to initialize insead of an
    // initializer. That way we don't have know the size of
    // matrix_memory at compile time.
    int matrix_memory[rows * cols];
    Matrix matrix = { matrix_memory, rows, cols };
    init_matrix(matrix);
    print_matrix(matrix);
    printf("\n");
    MatrixElement elt = {matrix, 1, 1};
    set_element(elt, 2);
    print_matrix(matrix);
}
```

```
0 0 0
0 0 0
0 0 0

0 0 0
0 2 0
0 0 0
```

1.6 Enum

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.

1.6.1 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, DOWN, LEFT, RIGHT
};
typedef enum direction Direction;

const char * const direction_names[N_DIRECTIONS] = {
    [UP] = "Up",
    [LEFT] = "Left",
    [DOWN] = "Down",
    [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 [3]
1 Down [1]
2 Left [2]
3 Up [0]
4 Right [3]
5 Down [1]
6 Left [2]
7 Up [0]
8 Right [3]
9 Down [1]
```

1.6.2 $\text{enum}_{\text{typedef.c}}$

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
```

1.6.3 EnumStart

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

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

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=101
STRAWBERRY=102
sizeof(Flavor)=4
```

1.6.4 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
```

1.6.5 $\text{Enum}_{\text{looptrick.c}}$

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

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

enum flavor {
    VANILLA,
    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;
            default:
                abort();
        }
    }
}

VANILLA=0
CHOCOLATE=1
STRAWBERRY=2
```

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

1.6.6 Enum Int

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

```
#include <stdio.h>
#include <stdlib.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];
//     if (flavor_name == NULL) {
//         printf("Flavor not found!\n");
//         abort();
//     }
    return flavor_name;
}



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