CMPSC 311 - Introduction to Systems Programming

Types, Structs, and Unions

Professors:

Sencun Zhu and Suman Saha

(Slides are mostly by *Professors Patrick McDaniel* and *Abutalib Aghayev*)

Types

- A data type is an abstraction that allows a programmer to treat different memory ranges as they were different classes (types) of data
  - e.g., integers, real numbers, text, records, etc.

```
// This is all just allocating memory with
// implicit/explicit sizes and values
short int si = 9;
long int li = 1234567890L;
float f = 3.14;
double d = 12324567890.1234567;
char c = 'a';
char *ptr = &si;
```

Insight: a variable name simple acts as an alias for a memory range.

Types

- The compiler uses type information in order to determine how to apply the logical operations defined by the code
  - Defines how different variables can be operated on, and ultimately what machine instructions are generated and executed

```
// Is this legal?
double one = 3.24, two = 4.5, res1;
int three = 3, four = 4059, res2;

// Are the ISA instructions for these two operations the same?
    res1 = one + two;
    res2 = three + four;
```

All programming languages use a *type system* to interpret code.

C Typing

PennState

- Programming languages are strongly or weakly "typed" (or some in between)
  - Such distinctions refer to the quality of the language in dealing with ambiguity in types and usage
  - C is weakly typed, which leads to great flexibility
    - Also leads to a lot of bugs ..
  - Q1: what value is output from the following code?
  - Q2: was that what the programmer intended?

```c
// How is "one" treated?
double one = 3.24;

if ( one / 3 == 1 ) {
      printf( "true" );
} else {
      printf( "false" );
}
```

Intuition: In general, a strongly typed language won't compile if variables of a different type are passed as parameters, compared, etc. Conversely, a weakly typed language (like C) will do its best to coerce (convert) it to be the correct type.

C Typing

- The code actually prints “false” because the 3 is turned into a floating point number (3.0) and the 1 is converted to (1.0).
  - Q2: was that what the programmer intended?

```
// How is “one” treated?
double one = 3.24;

if ( one / 3 == 1 ) {
      printf( “true” );
} else {
      printf( “false” );
}
```

Compiled expression:

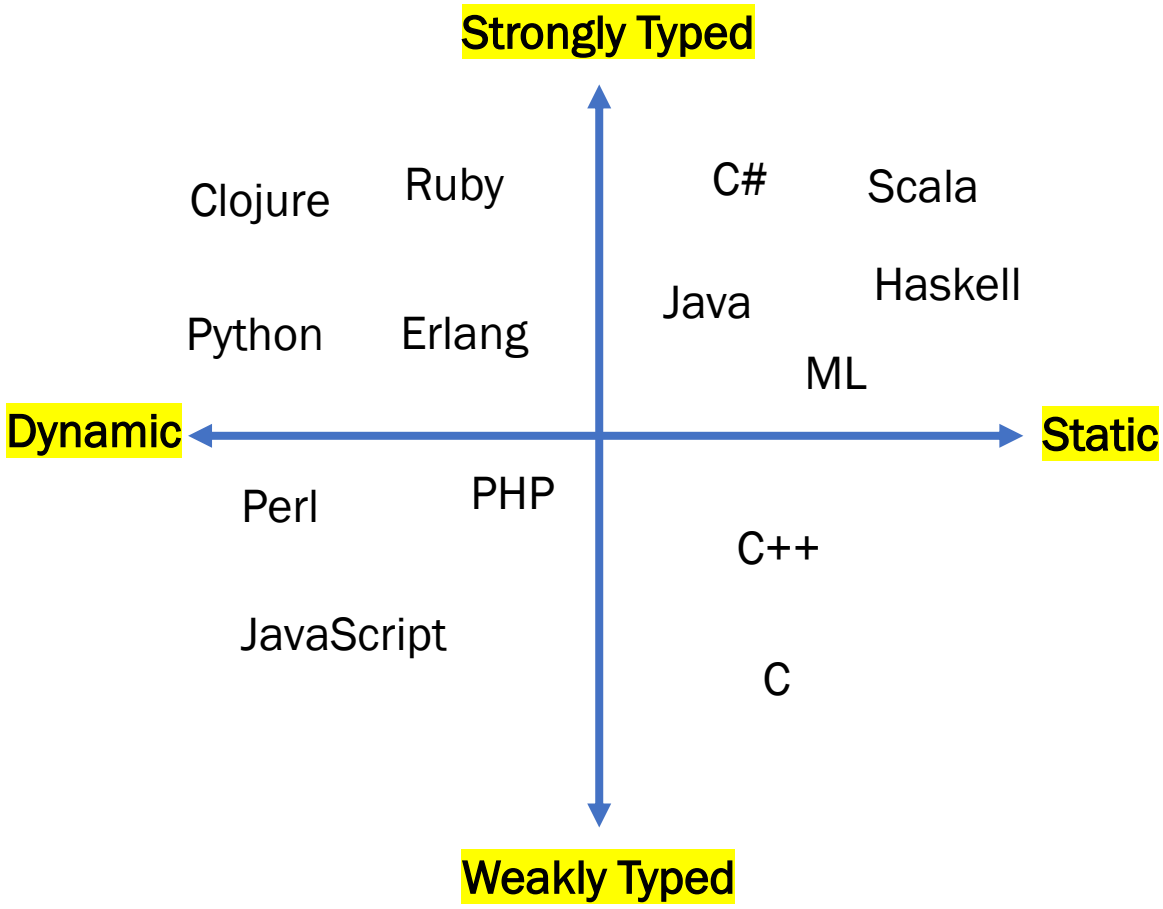
`(3.24/3.0) == (1.0)`

- You need to be careful in dealing with ..
  - Expressions (e.g., `one + 1`)
  - Parameter passing (e.g., `f(one)`)
  - Comparisons (e.g., `one == 1`)

Static vs Dynamic Typing

PennState

- No accepted definition of strong vs weak typing
- Another aspect of typing:
  - Static typing - this occurs when the type of data is decided at **compile time**, and type conversion occurs at that time
    - Examples: C, C++, Java, …
  - Dynamic typing - this occurs when the **run-time** environment dynamically applies types to variables as need
    - Examples: Perl, Python, Ruby
- Pros and cons of typing approaches



Type casting

- Q: But what if you want to control the way the type is converted?
- A: You can annotate a type conversion explicitly using type **casting**
- Syntax:

```
(type)variable
```

- where
  - type is a type to covert to
  - variable is the variable to be converted
  - by converting it `one` to an int, the semantics of the expression change to be all integers

```
// Now what?
double one = 3.24;

if ( (int)one / 3 == 1 ) {
      printf( "true" );
} else {
      printf( "false" );
}
```

Compiled expression:

```
(3/3) == (1)
```

Legal Type Casting

PennState

```c
int main(void)
{
        short int si = 9;
        long int li = 1234567890L;
        float f = 3.14;
        double d =  12324560.1234567;
        char c = 'a';
        char *ptr = &si;

        printf("short int %d %f %p\n", (int)si, (float)si, (char *)si);
        printf("long int %d %f %p\n", (int)li, (float)li, (char *)li);
        printf("float %d %f (ERR)\n", (int)f, (float)f);
        printf("double %d %f (ERR)\n", (int)d, (float)d);
        printf("char %d %f %p\n", (int)c, (float)c, (char *)c);
        printf("ptr %d (ERR) %p\n", (int)ptr, (char *)ptr);
        return 0;
}
```

```
mcdaniel@ubuntu:typecast$ ./typecast
short int 9 9.000000 0x9
long int 1234567890 1234567936.000000 0x499602d2
float 3 3.140000 (ERR)
double 12324560 12324560.000000 (ERR)
char 97 97.000000 0x61
ptr -716365630 (ERR) 0x7fffd54d20c2
```

Defining types: typedef

- The C `typedef` key word is a way of extending the C type system, i.e., to declare new new types for the compiler to recognize and use
- Syntax: `typedef [old type] [new type];`
- where
  - old type is a type definition suitable for declaration
  - new type is the type to be added
- Example:

```
typedef unsigned char bitfield;
```

Using user-defined types

PennState

- You can use the new type anywhere you use built in types:

```
// Type Declaration
typedef unsigned char bitfield;

...

// Return values and function parameters
bitfield myFunction( bitfield x, int y ) {

   // Local variables
   bitfield z;
   float a;
...
   // Type casting
   return( (bitfield)1 );
}
```

Note: the compiler treats the new type exactly as the old type (the new name acts simply as an *alias* for the original type)

Programming 101

- Q: When should you define your own types?
- A: you are working on a system that
  - will have a lot of code …
  - last a long time …
  - need to be ported …
  - have multiple revisions …
  - have a maintenance life time …
  - rely on standards …

- System-specific types afford you flexibility to alter the foundational data structures and types quickly and apply to large code bases.

Structures

- A structure is an organized unit of data that is treated as a single entity (variable)
  - Can be defined like any other variable
  - Have an implicit “type” (whether typedef-ed or not)
- Syntax

```
struct { [definition] } [variable(s)];
```

- Where
  - definition is the layout of the structure as list of variable definitions (these variable parts of the struct definition are called fields)
  - variable(s) are (one or more) variables which have the type structure

A Basic Example

```
// Vehicle structure
struct {
     char    name[128];        // Make and model
     int     mileage;          // The current mileage
} gremlin, cayman, cessna180, montauk;
```

Referencing fields in a C struct

PennState

- The period “ . ” is used to reference the fields of the struct

```
// Vehicle structure
struct {
    char    name[128];      // Make and model
    int     mileage;        // The current mileage
} gremlin, cayman, cessna180, montauk;

// Accessing the contents of the struct
strcpy(cayman.name, "My favorite car"); // strcpy will be covered later
cayman.mileage = 1240;
```

Enumerated types are useful …

- Recall that enum allows you to associate integer values with names
  - <ID?> can be anything
  - <value> must be integer

```
enum {
   <ID1> = <value>,
   <ID2> = <value>,
...
} variable;
```

Example:

```
enum {
   SUNDAY = 0,
   MONDAY = 1,
   TUESDAY = 2,
   WEDNESDAY = 3,
   THURSDAY = 4,
   FRIDAY = 5,
   SATURDAY = 6
} daysOfWeek;
```

Enumerated types are useful …

- Note that the <value> part of the declaration is optional
  - Compiler will assign integers to these values

```
enum {
   <ID1>,
   <ID2>,
...
} variable;
```

Example:

```
enum {
   SUNDAY,
   MONDAY,
   TUESDAY,
   WEDNESDAY,
   THURSDAY,
   FRIDAY,
   SATURDAY
} daysOfWeek;
```

Enumerated types are useful ...

PennState

- You can use the names in any place that you would use an integer.

```
enum {
   SUNDAY,
   MONDAY,
   TUESDAY,
   WEDNESDAY,
   THURSDAY,
   FRIDAY,
   SATURDAY
} daysOfWeek;
```

```
int x = MONDAY;
int y = TUESDAY * FRIDAY;

if (day == SATURDAY) {
...
}

int func(int x);
...
func(TUESDAY);
```

Structures can be complex

PennState

```
// Vehicle structure
struct {
   enum {
      AUTOMOTIVE     = 0,  // Automobile or equivalent
      AERONAUTICAL   = 1,  // Airplane, rotorcraft, ..
      MARINE         = 2,  // Boat or similar
   } type;
   char    name[128];      // Make and model
   int     milage;         // The current milage
} gremlin, cayman, cessna180, montauk;

// Example
cayman.type = AUTOMOTIVE;
if (cayman.type == AUTOMOTIVE) {
   printf("This is a car\n");
}
```

Structures can be nested

PennState

```
// Vehicle structure
struct {
  enum {
    AUTOMOTIVE   = 0,  // Automobile or equivalent
    AERONAUTICAL = 1,  // Airplane, rotorcraft, ..
    MARINE       = 2     // Boat or similar
  } type;

  char   name[128];      // Make and model
  int    milage;         // The current milage
  struct {
     int   cylinders;    // The number of cylinders
     int   horsepower;   // The total horsepower
     int   hours_smoh;   // Hours since last major overhaul
  } engine;              // Engine Specification/history
} gremlin, cayman, cessna180, montauk;

// Example
cayman.engine.cylinders = 6;
cayman.milage = 1240;
```

Unions

- A union is a way to overlay different data structures over the same memory region
  - selectively interpret the data in place
- Syntax `union { [definition] } [variable(s)];`
- Where
  - definition is the list of alternate data items
  - variable(s) are (one or more)

```
union {
   char   vin[17];         // Vehicle ID (car)
   char   tail_number[8]; // Tail number (airplane)
   char   hull_id[12];     // Hull ID (boat)
}  vehicle_id;             // The vehicle identifier
```

Unions

PennState

```
union {
    char    vin[17];         // Vehicle ID (car)
    char    tail_number[8]; // Tail number (airplane)
    char    hull_id[12];     // Hull ID (boat)
}  vehicle_id;               // The vehicle identifier

// Example
strcpy(vehicle_id.vin, "123456"); // strcpy will be covered later
printf("%s\n", vehicle_id.vin);  // prints "123456"
vehicle_id.tail_number = "F-BOZQ";
printf("%s\n", vehicle_id.vin);  // prints "F-BOZQ"
printf("Size:%lu\n", sizeof(vehicle_id)); // prints "17"
```

Bringing it together

PennState

- This is a really ugly structure …

```
// Vehicle structure
struct {
  enum {
     AUTOMOTIVE   = 0,  // Automobile or equivalent
     AERONAUTICAL = 1,  // Airplane, rotorcraft, ..
     MARINE           = 2   // Boat or similar
   } type;
   char   name[128];       // Make and model
   int    milage;          // The current milage
   struct {
      int   cylinders;     // The number of cylinders
      int   horsepower;    // The total horsepower
      int   hours_smoh;    // Hours since last major overhaul
   } engine;               // Engine Specification/history
   union {
      char   vin[17];         // Vehicle ID (car)
      char   tail_number[8]; // Tail number (airplane)
      char   hull_id[12];     // Hull ID (boat)
   } vehicle_id;              // The vehicle identifier
} gremlin, cayman, cessna180, montauk;
```

This is where types come in ...

PennState

```
// Define the vehicle information
typedef enum {
  AUTOMOTIVE       = 0,  // Automobile or equivalent
  AERONAUTICAL     = 1,  // Airplane, rotorcraft, ..
  MARINE           = 2   // Boat or similar
} VEHICLE_TYPE;

typedef struct {
  int   cylinders;     // The number of cylinders
  int   horsepower;    // The total horsepower
  int   hours_smoh;    // Hours since last major overhaul
} ENGINE_INFO;         // Engine specification/history

typedef union {
  char   vin[17];         // Vehicle ID (car)
  char   tail_number[8]; // Tail number (airplane)
  char   hull_id[12];     // Hull ID (boat)
} VEHICLE_IDENT;          // The vehicle identifier

// Vehicle structure
typedef struct {
    char          name[128];    // Make and model
    int           milage;       // The current milage
    VEHICLE_TYPE  type;         // The type of vehicle
    ENGINE_INFO   engine;       // Engine specification/history
    VEHICLE_IDENT vehicle_id;   // The vehicle identification
} VEHICLE;

// Now define the variables
VEHICLE gremlin, cayman, cessna180, montauk;
```

Accessing fields by pointer (dereferencing)

PennState

- When handling a pointer to a struct, the fields are accessed with the “->” operator instead of the “ . ”

```
VEHICLE cayman;
VEHICLE *vehicle = &cayman;
strcpy(vehicle->name, “2013 Porsche Cayman S”);
vehicle->engine.cylinders = 6;
```

Conditional Operator

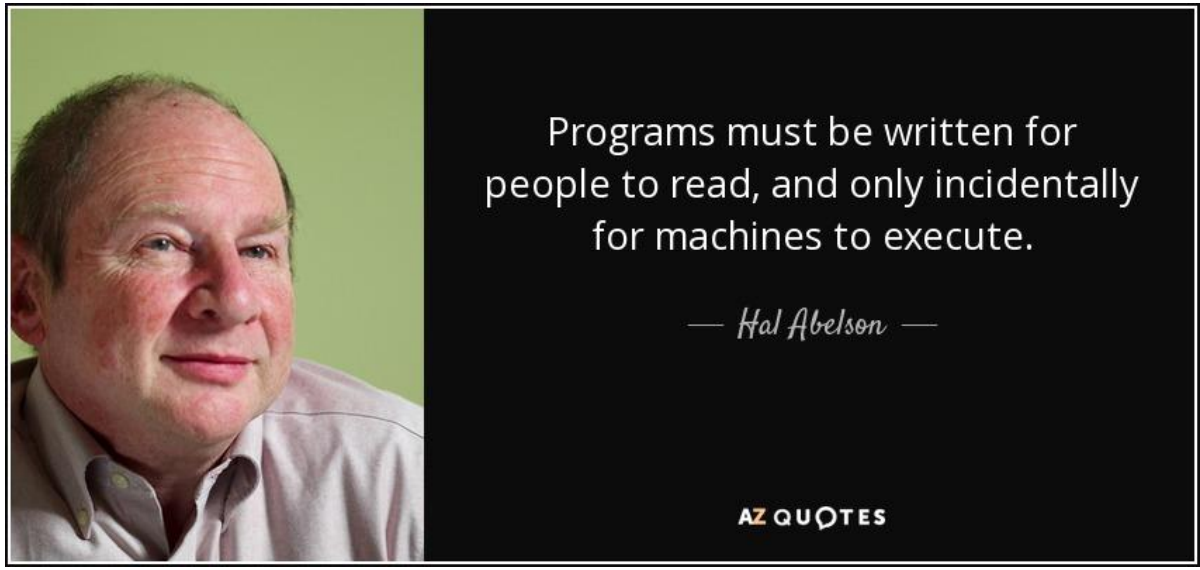
- Consists of two symbols “?” and “:” used as follows: `expr1 ? expr2 : expr3`
- Read as “if expr1 then expr2 else expr3”
  - expr1 is evaluated first
    - if its value isn’t zero, then the result is the evaluation of expr2
    - else the result is the evaluation of expr3

```
int i = 1, j = 2, k;
k = i > j ? i : j;
k = (i >= 0 ? i : 0) + j;
```

- Common usage in printf

```
if (i > j)
  printf("%d\n", i);
else
  printf("%d\n", i);

//or
printf("%d\n", i > j ? i : j);
```



Working with structs

PennState

- Other coding topics : what is going on with the multiline print string and “?” expressions?

```
VEHICLE *vehicle = &cayman;
printf( "*** Vehicle Information **\n"
        "Name         :   %s\n"
        "Milage       :   %u\n"
        "Vehicle type :   %s\n"
        "Cylinders    :   %u\n"
        "Horsepower   :   %u hp\n"
        "SMOH         :   %u hours\n"
        "VIN          :   %s\n",
        vehicle->name,
        vehicle->milage,
        (vehicle->type == AUTOMOTIVE) ? "car" :
                (vehicle->type == AERONAUTICAL) ? "airplane" : "boat",
        vehicle->engine.cylinders,
        vehicle->engine.horsepower,
        vehicle->engine.hours_smoh,
        (vehicle->type == AUTOMOTIVE) ?  vehicle->vehicle_id.vin :
                (vehicle->type == AERONAUTICAL) ?
                vehicle->vehicle_id.tail_number :
                vehicle->vehicle_id.hull_id );
```

Working with structs

PennState

- Other coding topics : what is going on with the multiline print string and “?” expressions?

```
VEHICLE *vehicle = &cayman;
printf( "*** Vehicle Information **\n"
        "Name          :   %s\n"
        "Milage        :   %u\n"
        "V
        "C
        "H
        "S
        "V
        ve
        ve
        (v
                                                    ane" : "boat",
        ve
        ve
        vehicle->engine.hours_smoh,
        (vehicle->type == AUTOMOTIVE) ?  vehicle->vehicle_id.vin :
                (vehicle->type == AERONAUTICAL) ?
                vehicle->vehicle_id.tail_number :
                vehicle->vehicle_id.hull_id );
```

```
Output:
*** Vehicle Information ***
Name          :   2013 Porsche Cayman S
Milage        :   1023
Vehicle type  :   car
Cylinders     :   6
Horsepower    :   325 hp
SMOH          :   100 hours
VIN           :   JH4TB2H26CC00000
```

How is the memory laid out?

PennState

```
#define MEM_OFFSET(a,b) ((unsigned long)&b)-((unsigned long)&a)
// Print out the values of the fields
printf( "                                SZ       Addr   Off\n" );
printf( "cayman                          %3lu  %p  0x%02lx\n",
        sizeof(cayman), &cayman, MEM_OFFSET(cayman,cayman) );
printf( "cayman.name                     %3lu  %p  0x%02lx\n",
   sizeof(cayman.name), &cayman.name, MEM_OFFSET(cayman,cayman.name) );
printf( "cayman.milage                   %3lu  %p  0x%02lx\n",
   sizeof(cayman.milage), &cayman.milage, MEM_OFFSET(cayman,cayman.milage) );
printf( "cayman.type                     %3lu  %p  0x%02lx\n",
   sizeof(cayman.type), &cayman.type, MEM_OFFSET(cayman,cayman.type) );
printf( "cayman.engine.cylinders         %3lu  %p  0x%02lx\n",
   sizeof(cayman.engine.cylinders), &cayman.engine.cylinders,
   MEM_OFFSET(cayman,cayman.engine.cylinders) );
printf( "cayman.engine.horsepower        %3lu  %p  0x%02lx\n",
   sizeof(cayman.engine.horsepower), &cayman.engine.horsepower,
   MEM_OFFSET(cayman,cayman.engine.horsepower) );
printf( "cayman.engine.hours_smoh        %3lu  %p  0x%02lx\n",
   sizeof(cayman.engine.hours_smoh), &cayman.engine.hours_smoh,
   MEM_OFFSET(cayman,cayman.engine.hours_smoh) );
printf( "cayman.vehicle_id.vin           %3lu  %p  0x%02lx\n",
   sizeof(cayman.vehicle_id.vin), &cayman.vehicle_id.vin,
   MEM_OFFSET(cayman,cayman.vehicle_id.vin) );
printf( "cayman.vehicle_id.tail_number   %3lu  %p  0x%02lx\n",
   sizeof(cayman.vehicle_id.tail_number), &cayman.vehicle_id.tail_number,
   MEM_OFFSET(cayman,cayman.vehicle_id.tail_number) );
printf( "cayman.vehicle_id.hull_id       %3lu  %p  0x%02lx\n",
   sizeof(cayman.vehicle_id.hull_id), &cayman.vehicle_id.hull_id,
   MEM_OFFSET(cayman,cayman.vehicle_id.hull_id) );
```

How is the memory laid out?

PennState

```
#define MEM_OFFSET(a,b) ((unsigned long)&b)-((unsigned long)&a)
// Print out the values of the fields
printf( "                                SZ      Addr   Off\n" );
printf( "cayman                         %3lu  %p  0x%02lx\n",
        sizeof(cayman), &cayman, MEM_OFFSET(cayman,cayman) );
printf( "cayman.name                    %3lu  %p  0x%02lx\n",
pri
pri
pri
pri
pri
pri
   sizeof(cayman.vehicle_id.vin), &cayman.vehicle_id.vin,
   MEM_OFFSET(cayman,cayman.vehicle_id.vin) );
printf( "cayman.vehicle_id.tail_number  %3lu  %p  0x%02lx\n",
   sizeof(cayman.vehicle_id.tail_number), &cayman.vehicle_id.tail_number,
   MEM_OFFSET(cayman,cayman.vehicle_id.tail_number) );
printf( "cayman.vehicle_id.hull_id      %3lu  %p  0x%02lx\n",
   sizeof(cayman.vehicle_id.hull_id), &cayman.vehicle_id.hull_id,
   MEM_OFFSET(cayman,cayman.vehicle_id.hull_id) );
```

Output:

	SZ	Addr	Off
cayman	160	0x601080	0x00
cayman.name	128	0x601080	0x00
cayman.milage	4	0x601100	0x80
cayman.type	4	0x601104	0x84
cayman.engine.cylinders	1	0x601108	0x88
cayman.engine.horsepower	2	0x60110a	0x8a
cayman.engine.hours_smoh	2	0x60110c	0x8c
cayman.vehicle_id.vin	17	0x60110e	0x8e
cayman.vehicle_id.tail_number	8	0x60110e	0x8e
cayman.vehicle_id.hull_id	12	0x60110e	0x8e

Lets take a closer look

- Lets add up the sizes of the variables
  - 128+4+4+1+2+2+max(17,8,12) = 158, not 160!
  - What happened?

```
Output:
                                SZ      Addr   Off
cayman                         160  0x601080  0x00
cayman.name                    128  0x601080  0x00
cayman.milage                    4  0x601100  0x80
cayman.type                      4  0x601104  0x84
cayman.engine.cylinders          1  0x601108  0x88
cayman.engine.horsepower         2  0x60110a  0x8a
cayman.engine.hours_smoh         2  0x60110c  0x8c
cayman.vehicle_id.vin           17  0x60110e  0x8e
cayman.vehicle_id.tail_number    8  0x60110e  0x8e
cayman.vehicle_id.hull_id       12  0x60110e  0x8e
```

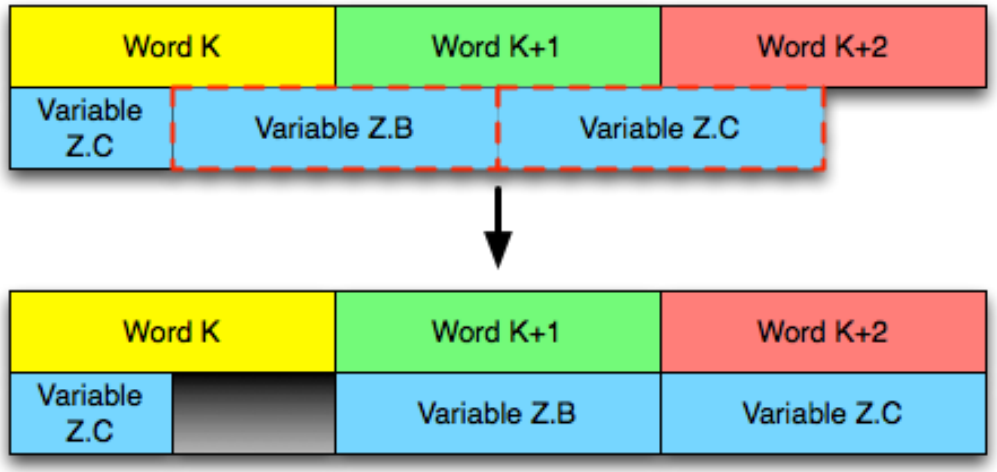
The conundrum …

- OK, lets do some computer science math …

```
0x00 + 128 = 128 (0x80)
0x80 + 4 = 132 (0x84)
0x84 + 4 = 136 (0x88)
0x88 + 1 = 137 (0x89) ... skipped a byte?
0x8a + 2 = 140 (0x8c)
0x8c + 2 = 142 (0x8e)
0x8e + 17 = 159 (0x9f) ... skipped a byte?
160?
```

The answer

- The compiler is “padding” your structure with unused memory to make sure that number aligns with a multiple of the machine word size.
  - It does this because many ISA instructions require the operable address to loadable from a word location



Note: the way a compiler pads is dependent on the underlying processor architecture. Beware when working with data from other computers.

Copy by assignment

- You can assign the value of a `struct` from a struct of the same type; this copies the entire contents

```c
#include <stdio.h>

struct Point {
  float x, y;
};

int main(int argc, char **argv) {
  struct Point p1 = {0.0, 2.0};
  struct Point p2 = {4.0, 6.0};

  printf("p1: {%f,%f}  p2: {%f,%f}\n", p1.x, p1.y, p2.x, p2.y);
  p2 = p1;
  printf("p1: {%f,%f}  p2: {%f,%f}\n", p1.x, p1.y, p2.x, p2.y);
  return 0;
}
```

```
p1: {0.000000,2.000000}  p2: {4.000000,6.000000}
p1: {0.000000,2.000000}  p2: {0.000000,2.000000}
```

You can return structs

PennState

```c
// a complex number is a + bi
typedef struct complex_st {
  double real;  // real component (i.e., a)
  double imag;  // imaginary component (i.e., b)
} Complex, *ComplexPtr;

Complex AddComplex(Complex x, Complex y) {
  Complex retval;

  retval.real = x.real + y.real;
  retval.imag = x.imag + y.imag;
  return retval;  // returns a copy of retval
}

Complex MultiplyComplex(Complex x, Complex y) {
  Complex retval;

  retval.real = (x.real * y.real) - (x.imag * y.imag);
  retval.imag = (x.imag * y.real) - (x.real * y.imag);
  return retval;
}
```

Bit fields

- Create numeric (integer) values that have a very specific width (in bits)
  - C supports this by identifying the bit width in declarations of integer fields, e.g.,

```
// Define the structure of bit fields
struct vehicle_props {
        uint32_t  registered : 1;
        uint32_t  color_code : 8;
        uint32_t  doors : 3;
        uint32_t  year : 16;
} props;

...

// Using the fields
props.registered = 1;
props.color_code = 14;
props.doors = 2;
props.doors = 9; // Legal, but out of range
props.year = 2013;
printf("Size of props: %lu\n", sizeof(props));   //Prints "4"
```