Types, Structs, and Unions

Devin J. Pohly <djpohly@cse.psu.edu>

Types

- A *data type* is just an abstraction
  - Allows a programmer to treat a memory range as if it were a certain kind (type) of data
  - e.g., integers, real numbers, strings of characters, records, etc.
- Note: a variable name simply acts as an alias for a memory range.

```
// All this does is allocate 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 = &c;
```

Types

- The compiler uses types to determine what exactly the code does
  - How different variables can be operated on
  - Ultimately what machine instructions are generated and executed
- All programming languages use a *type system* to interpret code.

```
// 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;
```

C typing

- Programming languages are often classified as *strongly* or *weakly typed*
  - Or somewhere in between...
  - Such distinctions refer to how the language deals with *ambiguity* in types and usage
  - C is weakly typed, which leads to great flexibility
    - … and the potential for great bugs

C typing

- What value is output from the following code?
- Was that what the programmer intended?

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

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

Static vs. dynamic typing

PENNSTATE 1855

- There is no clear, widely-accepted definition of strong and weak typing
- Another way to look at typing:
  - *Static typing*: the type of data is decided at compile time, and type conversion occurs at that time
    - Examples: C, C++, Java
  - *Dynamic typing*: the run-time environment dynamically applies types to variables as needed
    - Examples: Perl, Python, Ruby, Objective-C

Type coercion

- Often you want to change the type of a variable
- *Type coercion* relies on the language to do it automatically
- Coercion: the integer literals 3 and 1 are turned into double expressions, so the output is "false". Was that the intent?
- What if you want to control the way the type is converted?

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

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

Type casting

- You can effect a temporary type conversion explicitly using *type casting*:

  `(type) expr`

  where `type` is the type to convert to, and `expr` is the variable or expression to convert

```
// How is it treated now?
double one = 3.24;

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

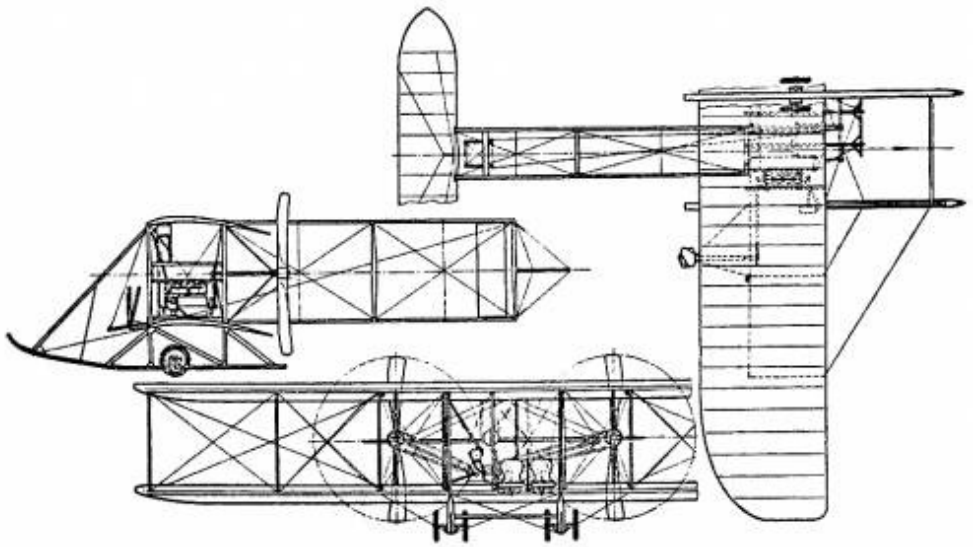
Legal type casting

```c
int main(int argc, char *argv[]) {
    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;
}
```

```
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
```

Structure types

- A struct is an organized set of data that are treated as a unit
  - I.e., it behaves like a single variable
  - Can be declared like any other variable
  - Type name includes the keyword `struct` and a tag you provide



Struct syntax and use

```
// Define the struct
struct myname {
  type1 field1;
  type2 field2;
  ...
};

// Declare/allocate a struct variable
struct myname foo;

// Access a field
x = foo.field1;
foo.field1 = 42;
```

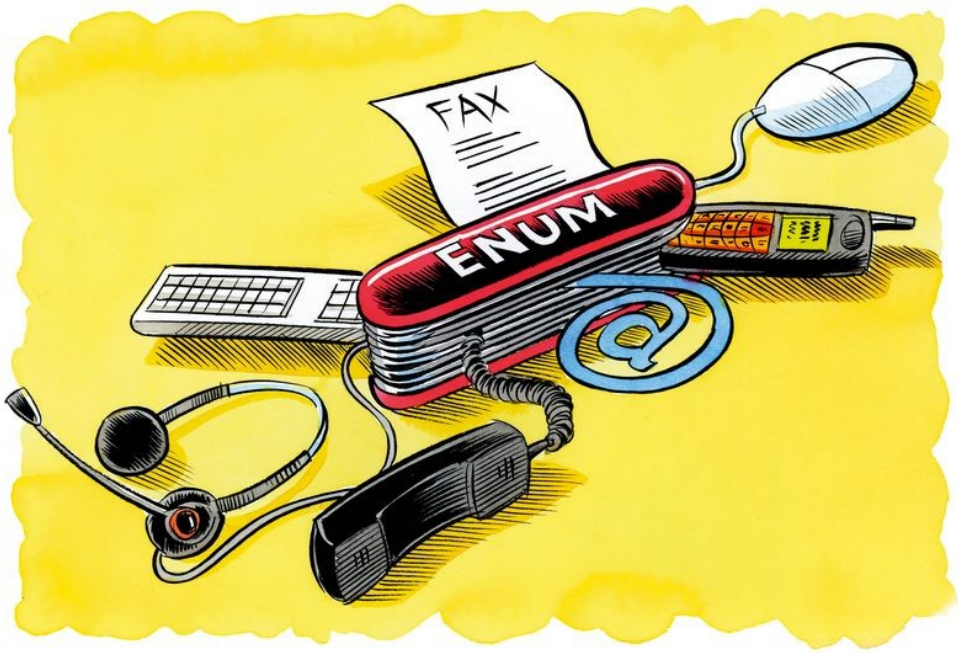
- Here, `typeX` is the type of each piece of data (*field*) inside this struct, and `fieldX` is the name of that field
  - Note the semicolon after the braces! This is a statement that defines a new type.

Basic example

```
struct vehicle {
    // Make and model
    char name[128];

    // Current mileage
    int mileage;
};

struct vehicle
    cayman, gremlin,
    cessna180, montauk;

...

montauk.mileage = 5500;

printf("%s: %d mi.\n",
    gremlin.name,
    gremlin.mileage);
```

Enumerations

- Allows you to treat a set of names as an efficient integer value
  - Used when there is a discrete range of values for a type that isn't already numeric
  - Type name includes the keyword enum and a tag



Enumeration syntax

```
enum vehicle_type {
  AUTOMOTIVE,
  AERONAUTICAL,
  MARINE
};

enum vehicle_type t;

t = MARINE;
```

- Note again the semicolon!
- You can also assign specific values if needed, e.g.:

```
SPECIAL = 999
```

Nested types

- Structs can include fields of any known type
  - Primitive types like int or float
  - Arrays
  - Pointers
  - Other structs!
  - Enumerations
  - etc.

```
struct engine_specs {
  int cylinders;
  int horsepower;
};

enum vehicle_type {
  AUTOMOTIVE,
  AERONAUTICAL,
  MARINE
};

struct vehicle {
    // Make and model
    char name[128];
    // Current mileage
    int mileage;
    // Type of vehicle
    enum vehicle_type type;
    // Engine specifications
    struct engine_specs engine;
};
```

Unions

- A union allows you to use the *same memory region* for several variables
  - Can even be of different types – **DANGER!**
    - C will not stop you from writing bits as one type and reading them as a different one!
  - If you ever use a union, you should consider its members to be *mutually exclusive*!

Unions

```c
// Define the union
union vehicle_ident {
  char vin[17];
  char tail_number[8];
  char hull_id[12];
};

// Declare/allocate a union variable
union vehicle_ident id;

// Access a member
printf("VIN: %s\n", id.vin);
```

- Defined just like a struct
  - Instead of containing all of the fields, it contains one of the fields
    - What type it treats the memory as depends on which member you are accessing
  - Only needs to be as big as the largest member

Bringing it all together

```c
struct engine_specs {
    int cylinders;
    int horsepower;
};

enum vehicle_type {
    AUTOMOTIVE,
    AERONAUTICAL,
    MARINE
};

union vehicle_ident {
    char vin[17];
    char tail_number[8];
    char hull_id[12];
};

struct vehicle {
    char name[128];
    int mileage;
    enum vehicle_type type;
    struct engine_specs engine;
    union vehicle_ident id;
};

struct vehicle cayman, gremlin, cessna180, montauk;
```

Type aliases: typedef

- Type names sometimes get long and unwieldy
- The C typedef statement creates an alias for an existing type:

```
typedef oldtype newtype;
```

- where
  - *oldtype* is a type name, written exactly as if you were declaring a variable
  - *newtype* is the new alias for this type
  - Convention: type aliases frequently end in _t
- Example:

```
typedef struct engine_specs engine_t;
```

Using user-defined types

- You can use the new type anywhere you use built-in types
  - Provides an abstraction
- Note: the compiler treats the alias exactly as if it were the original type

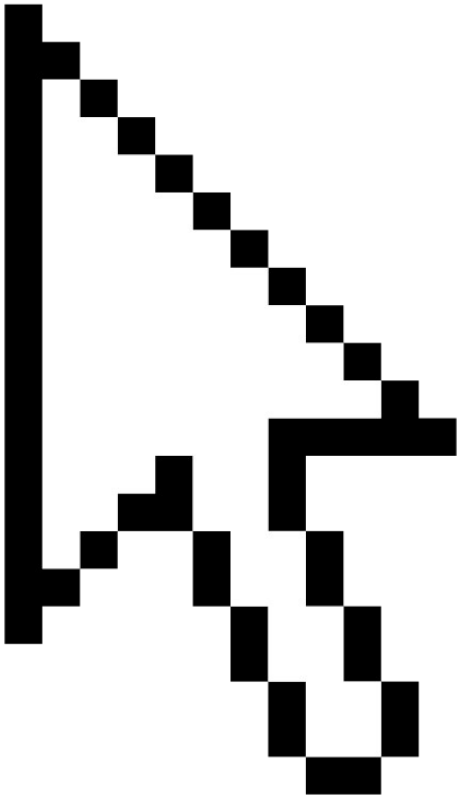
```
// Type declaration
typedef unsigned char age_t;

// Return values and function parameters
age_t myFunction(age_t x, int y) {
  // Local variables
  age_t z;
  float a;

  // Type casting
  return (age_t) 1;
}
```

Struct pointers

- Work exactly like any other pointer

```
struct vehicle *pv = &cayman;
```

- How could you get the `mileage` field given this pointer?



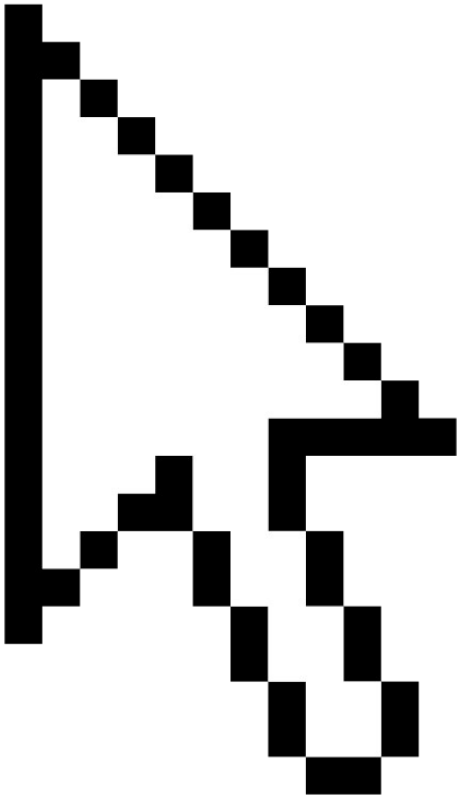
Struct pointers

- Work exactly like any other pointer

```
struct vehicle *pv = &cayman;
```

- How could you get the `mileage` field given this pointer?

```
int m = (*pv).mileage;
```



Struct pointers

- Work exactly like any other pointer

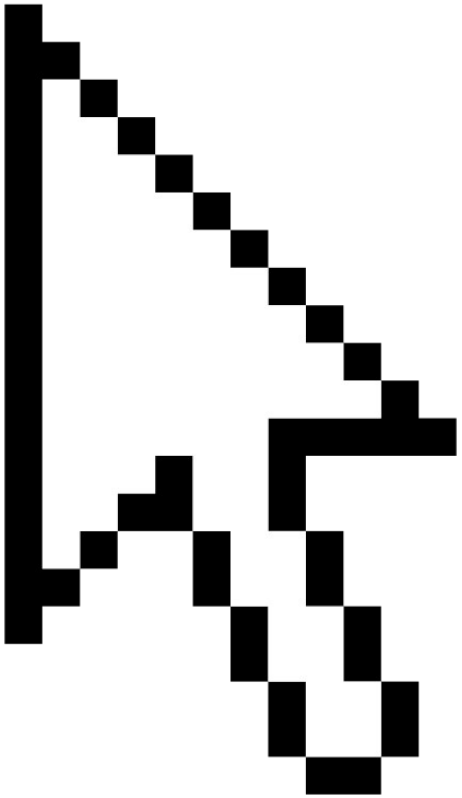
```
struct vehicle *pv = &cayman;
```

- How could you get the `mileage` field given this pointer?

```
int m = (*pv).mileage;
```

  - So common that it has its own syntactic sugar: the arrow operator.

```
int m = pv->mileage;
```



Struct layout in memory

```
#define MEM_OFFSET(a, b) (((uintptr_t) &b) - ((uintptr_t) &a))

// Print out the addresses of the fields
printf("                                SZ    Addr          Ofs\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.mileage                 %3lu %p 0x%02lx\n",
    sizeof(cayman.mileage), &cayman.mileage, MEM_OFFSET(cayman, cayman.mileage));
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.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));
```

Struct layout in memory

```
#define MEM_OFFSET(a, b) (((uintptr_t) &b) - ((uintptr_t) &a))

// Print out the addresses of the fields
printf("                                  SZ    Addr          Ofs\n");
printf("cayman                           %3lu %p 0x%02lx\n",
    sizeof(cayman), &cayman, MEM_OFFSET(cayman, cayman));
printf("caym
    sizeof(c                                                    ));
printf("caym
    sizeof(c                                                    .mileage));
printf("caym
     sizeof(                                                    e));
printf("caym
    sizeof(c
    MEM_OFFS
printf("caym
    sizeof(c
    MEM_OFFS
printf("caym
    sizeof(c
    MEM_OFFS
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));
```

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

Wait a second...

- Add up the sizes of the variables:
  - 128 + 4 + 4 + 1 + 2 + max(17, 8, 12) = 156
  - But `sizeof(cayman)` said 160!
  - What happened?

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

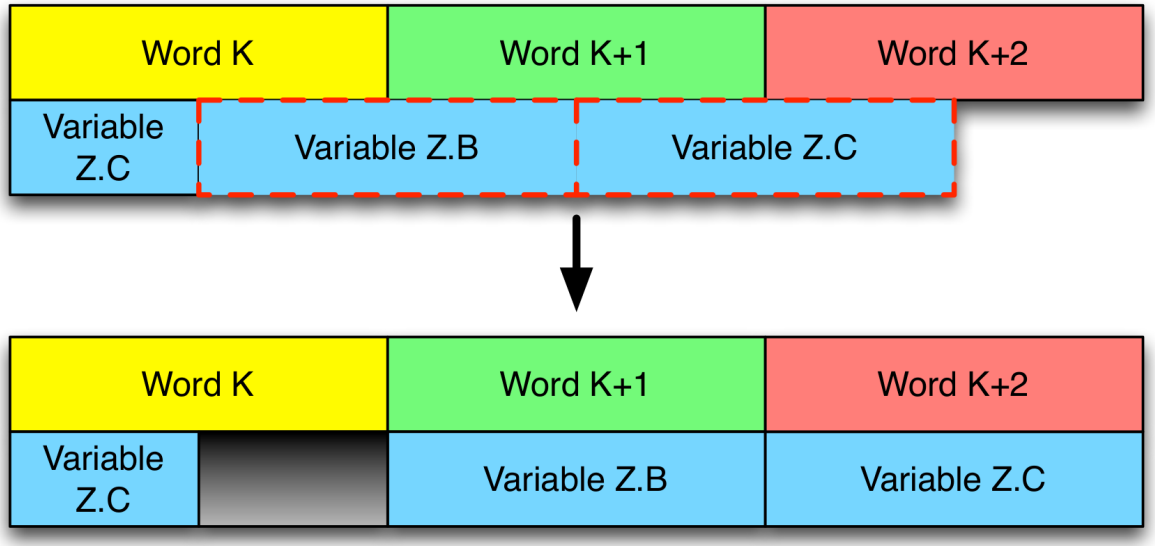
Let's take a closer look

- If we do a little hex math...
  - From offset 0x88 to 0x8a is 2 bytes (skipped 1)
  - From offset 0x8c to 160 is 20 bytes (skipped 3)

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

The answer

- The compiler may "pad" the structure with unused memory so that offsets align with a multiple of the machine word size.
  - This is because many ISAs have instructions that require the target address(es) to be "word-aligned"
  - **Caution**: the way this works varies between architectures and compilers. Beware when working with data from other computers (network communication, file formats, etc.)!



Bit fields

- Often you want to create numeric (integer) fields that have a very specific width (in bits)
  - C supports this by explicitly identifying the bit width in declarations of integer fields:

```
// Define a structure of bit fields
struct vehicle_props {
    uint16_t registered: 1;
    uint16_t color_code: 8;
    uint16_t doors: 3;
    uint16_t year: 16;
};

struct vehicle_props props;

// Use the fields
props.registered = 1;
props.color_code = 14;
props.doors = 2;
props.doors = 9;  // Legal, but out of range!
props.year = 2014;
```

eof(1)