CSE 351 Lecture 14 – Structs & Alignment

Structs and Typedef in C

```
struct struct_tag {
  type_1 field_1;
  ...
  type_N field_N;
};
```

A ***struct*** in C is a user-defined, structured group of variables.
A struct definition is formatted as shown on the right:

The user-chosen ***struct tag*** is part of the name of the new data type we are defining, which is `struct struct_tag` (two-part name). The struct tag can be omitted if you don't need to use this data type name elsewhere in your code or you immediately typedef it. The struct name can only be used in code *after* the struct definition (and when it is in scope). Typically, a struct definition will appear like the example above at the top of a file in the global scope, but it is possible to combine a struct definition with variable declarations (and initialization). Not recommended, but you may encounter code that looks like the following on occasion:

```
struct {int x; int y;} var;   # Declares a variable var of an unnamed struct
                              # type that contains two integers x and y.

struct point {                # Defines the type struct point that contains
  int x;                      # two integers x and y, but also declares an
  int y;                      # instance of the struct (pt) and a pointer to
} pt, *ptr;                   # struct point (ptr).
```

The purpose of the struct definition is so that your compiler/program knows the *size* and *layout* of an instance of the struct. A struct holds an arbitrary collection of ***struct fields*** of different variable types. The only exception is that a struct cannot hold an instance of itself – how much space would be allocated for it? – but a pointer of the struct type is okay. Fields are accessed from an instance using the '`.`' operator (*e.g.* `pt.x`) or from a pointer by either (1) dereferencing and using '`.`' or (2) using the '`->`' operator (*i.e.* `ptr->y` is equivalent to `(*ptr).y`).

The two-word struct data type names (`struct struct_tag`) are a bit cumbersome so we often combine them with **`typedef`**, which allows you to create aliases to other data types such as:

```
typedef unsigned int uint;  # typedef <data type> <alias>;
```

For structs, a typedef statement can be used *after* or *combined with* the struct definition to make a more manageable data type:

```
# typedef after definition
struct point_st {
  int x;
  int y;
};
typedef struct point_st Point;
Point pt1;
```

```
# typedef combined with definition
typedef struct {   # tag now
  int x;           # optional
  int y;
} Point;

Point pt1;
```

Alignment

The size and layout of a struct instance in C is determined by (1) the user-defined ordering of the struct fields and (2) alignment requirements. As a reminder, a primitive object of size K bytes in memory is considered ***aligned*** if its address is a multiple of K.

The layout of the struct in memory is determined by the compiler. It will follow the ordering of the fields in the definition of the struct but insert padding between fields to make sure that each individual field is aligned. For example, if the struct (theoretically) started at address 0x0, and the first two fields are a `short` followed by an `int`, then the `short` would be placed at address 0x0 (multiple of 2), but there would be 2 bytes of padding before the `int` would be placed at address 0x4 (multiple of 4). This unused space (padding) *between fields* is known as ***internal fragmentation***.

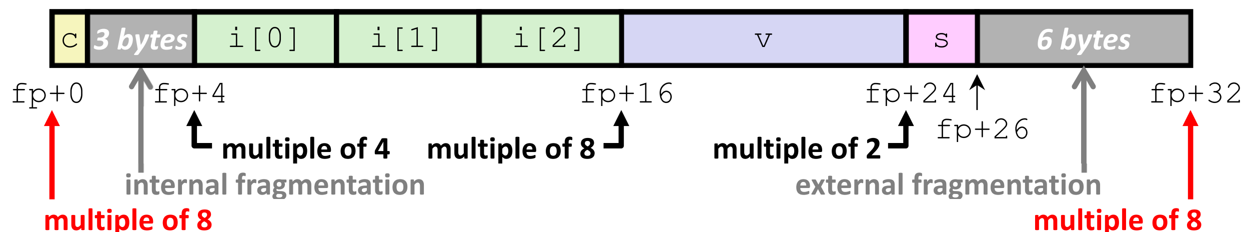
> **Note:** For arrays, notice that the contiguous allocation guarantees that each element will be K bytes after the previous one. This means that as long as the array starts at a multiple of K, *every* element will be properly aligned, so the alignment requirement for an array is the size of an individual element and *not* the size of the whole array (*i.e* K instead of $n \times K$).

The overall size of a struct is also subject to alignment requirements. This is to guarantee that the individual fields are properly aligned (*e.g.* we relied on starting at 0x0 in the example above) and that consecutively-allocated structs (*i.e.* in an array) also are properly aligned. The alignment requirement on the overall size of the struct is the alignment requirement of its *largest field*, *i.e.* $K_{\text{max}} = \max(K_i)$ for all fields in the struct. After placing each field of the struct, additional padding is added, if necessary, to the end of the struct to make the size of the struct a multiple of K_{max}. This unused space (padding) *between struct instances* is known as ***external fragmentation***.

```
struct frag {
  char   c;
  int    i[3];
  struct frag *p;
  short  s;
} f, *fp = &f;
```

As a more complete example, take the following struct:
The field alignment requirements are: $K_{\text{c}} = 1$, $K_{\text{i}} = 4$, $K_{\text{p}} = 8$, $K_{\text{s}} = 2$.
The overall struct alignment requirement is $K_{\text{max}} = 8$ (from K_{p}).

This means that the address of `f` (the value stored in `fp`) will be a multiple of 8. Then `c` will be properly aligned at `fp+0` and 3 bytes of internal fragmentation will be added so that `i` starts at `fp+4`, which is a multiple of 4. After 12 bytes of the array `i`, `p` can be placed at `fp+16` without any padding because it is a multiple of 8. `s` can be placed at `fp+24`, which brings the current end of the struct to `fp+26`. Since we need the size to be a multiple of 8, we add 6 bytes of external fragmentation to bring the final struct size to 32 bytes:



By clever choice of field ordering, sometimes a programmer can reduce the size of each struct instance!