User Defined Types in C

The C Programming Language: Chapter 6

Types in C

- Built-in types: char, float, int, double, short, …
- Modified types
  - Arrays: char ____[5], int _____[3][7], …
  - Pointers: int *___, char *____, …

- User Named Type Extensions
  - Structures
  - Unions
  - Enumerations
- User Defined Types - typedef

User Named Type Extensions

- Type consists of <keyword> <tag>
  - e.g. “struct llnode” or “union data” or “enum colors”
- Type must be defined: <type> { <definition> } …
  - e.g. “enum colors { red, green, blue }”
- … before it is used to declare a variable: <type> <variable>;
  - e.g. “enum colors paintColor;”
- Type definition and variable declaration may occur in the same statement: “enum colors { red, green, blue } paintColor;”
- If type is used only once, name is not required… “anonymous” type
  - enum { up, down } elevator_motion;

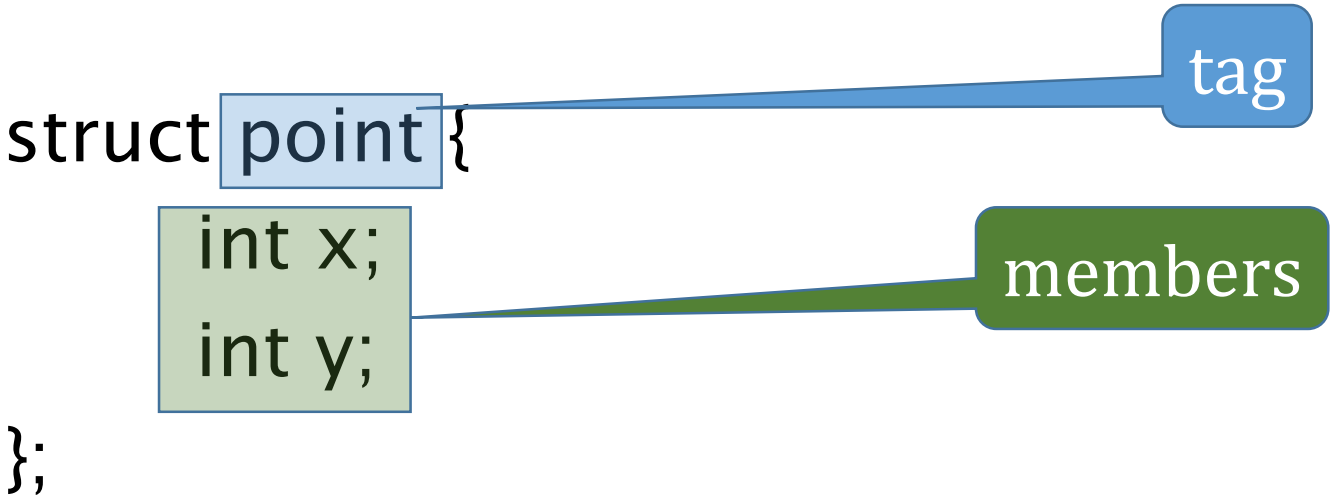
What is a Structure?

- A method to collect a group of variables
  - Treat the group as a single entity
  - Enable access to each variable within the group
  - Pass the whole group around

struct [<tag>] { <members> } [<instance>];

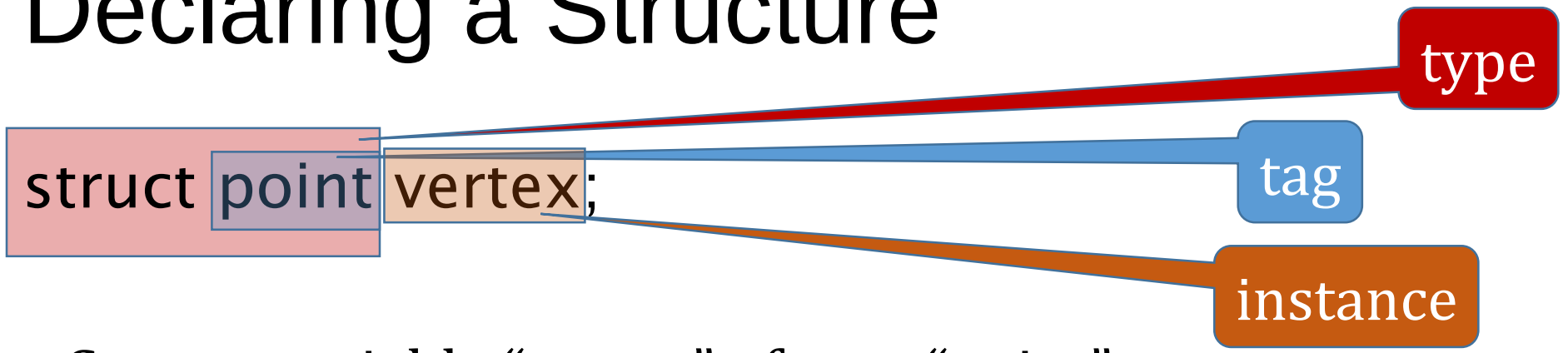
See also:
https://en.wikipedia.org/wiki/Struct_(C_programming_language)

Defining a Structure



- Creates a new data type: "struct point"
  - Two sub-fields: "x" and "y", both integers
- Does NOT create any variables with this data type!
- Does NOT reserve any memory

Declaring a Structure



- Creates variable "vertex" of type "point"
- "point" must be defined previously
- Allocates memory for all sub-fields of vertex
- "vertex" has two sub-fields
  - vertex.x
  - vertex.y

Label	Address	Value
	…	
vertex.y	0xc70C	0x0004
vertex.x	0xc708	0x0003
	…	

Example of an anonymous structure

```
struct {
    int x;
    int y;
} vertex;
```

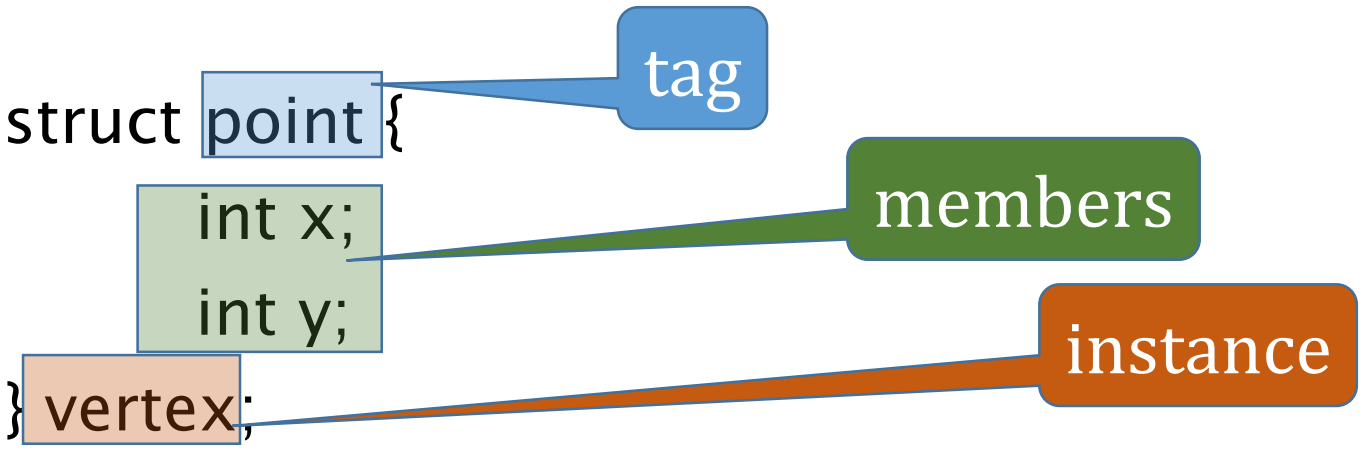
members

instance

Label	Address	Value
	…	
vertex.y	0xa70C	0x0004
vertex.x	0xa708	0x0003
	…	

- Creates variable “vertex” of type “anonymous struct”
- Allocates memory for all sub-fields of vertex
- “vertex” has two sub-fields
  - vertex.x
  - vertex.y

Example definition and declaration



Label	Address	Value
	…	
vertex.y	0x970C	0x0004
vertex.x	0x9708	0x0003
	…	

- Creates a new data type: “struct point”
- Creates variable “vertex” of type “struct point”
- Allocates memory for all sub-fields of vertex
- New instances of type “struct point” can be created

What can I do with a Structure?

```
struct point origin = {0,0};
struct point here;
struct point moveX(struct point from,int dist) {
        from.x+=dist;
        return from;
}
here=origin;
here=moveX(point,3);
```

initialize

pass as argument

return

assign

Data Alignment

- Data alignment: the address of a variable should be divisible by the size of the variable
  - char starts at an address divisible by 1
  - short starts at an address divisible by 2
  - int starts at an address divisible by 4
  - …
- On most hardware, memory fetch occurs faster when data is aligned
- In most cases, compiler takes care of alignment for us
- See https://en.wikipedia.org/wiki/Data_structure_alignment

Structure Data Layout

- Typically, members of structures are in contiguous memory
- However, when data is aligned, compiler may insert padding
- Compiler may insert padding at the end to ensure entire structure is aligned

Structure Padding Example

How you code it

```
struct student {
    char fist_init;
    char * last_name;
    short age;
    int id;
    char grade;
} class[80];
```

How it really is

```
struct student {
        char first_init;
        char pad1[3];
        char * last_name;
        short age;
        char pad2[2];
        int id;
        char grade;
        char pad3[3]
} class[80];
```

use sizeof(struct student) to determine how many bytes are needed

Structure Definition to avoid padding

How you code it

```
struct student {
        char * last_name;
        int id;
        short age;
        char fist_init;
        char grade;
} class[80];
```

How it really is

```
struct student {
        char * last_name;
        int id;
        short age;
        char fist_init;
        char grade;
} class[80];
```

Structure Pointers

```
struct xys { int x; int y}; // Create new type – struct xys
struct xys * vp = (struct xyx *)malloc(sizeof(struct xys));
vp->x=5;
vp->y=3;
```

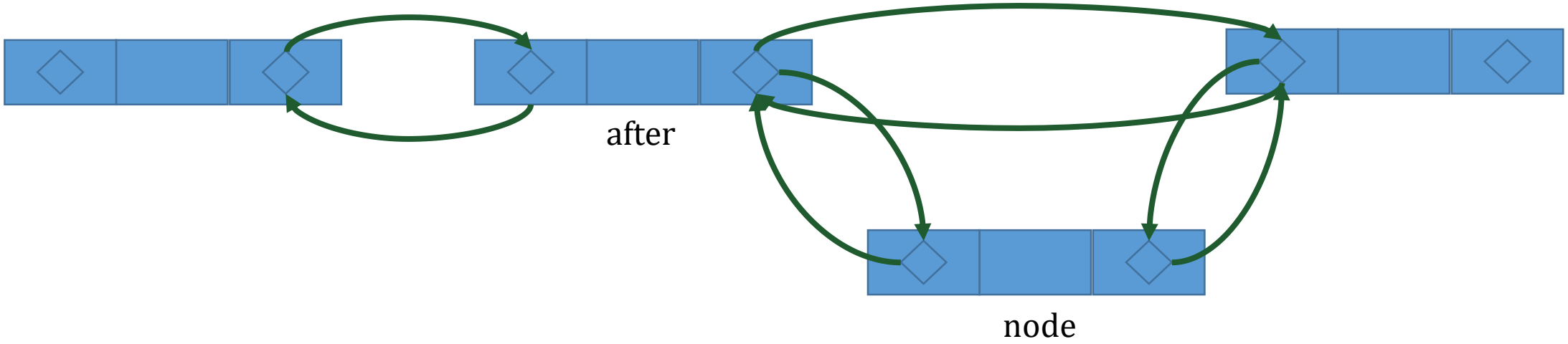
- Note: sizeof(struct xys) returns total size of all sub-fields
- "vp" is a pointer to an xys structure
- Access to sub-fields using "->" arrow
  - vp->x is shorthand for (*vp).x

Label	Address	Value
vp	0x7c04	0x7708
	…	
vp->y	0x770C	0x0003
vp->x	0x7708	0x0005
	…	

Self-referential Structures

```
struct dl_list {
        struct dl_list * prev;
        int value;
        struct dl_list * next;
} root;
```

```
void insert(dl_list * node, dl_list *after) {
        node->next=after->next;
        node->next->prev=node;
        node->prev=after;
        after->next=node;
}
```



Handling NULL start/end of list

```
void insert(dl_list * node, dl_list *after) {
        if (after==NULL) { // Insert before root
                node->next=root;
                root=node;
        }
        else node->next=after->next;
        if (node->next!=NULL && node->next->prev!=NULL)
                node->next->prev=node;
        node->prev=after;
        if (after!=NULL) after->next=node;
}
```

Other data structures

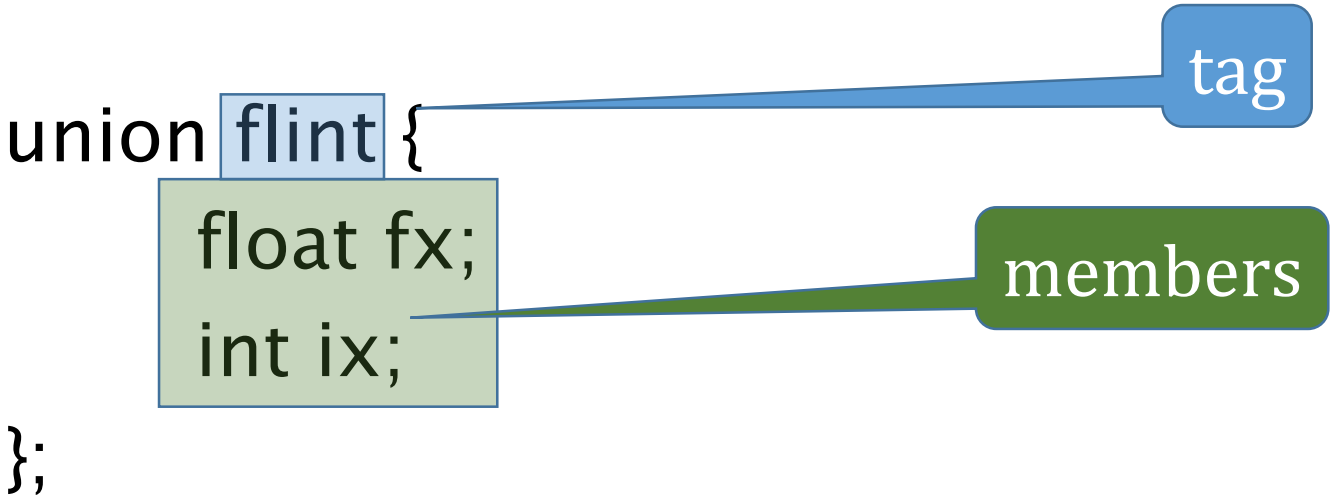
How can we use self-referential structures to create:

- Circular singly linked lists
- Binary trees with left and right children
- Binary trees with left and right children and parent pointers
- Arbitrary graphs with N nodes and E edges

What is a Union

- Nearly a structure but
- Fields overlap (start at same memory location)
- Enable multiple types for the same data

Union as a Type

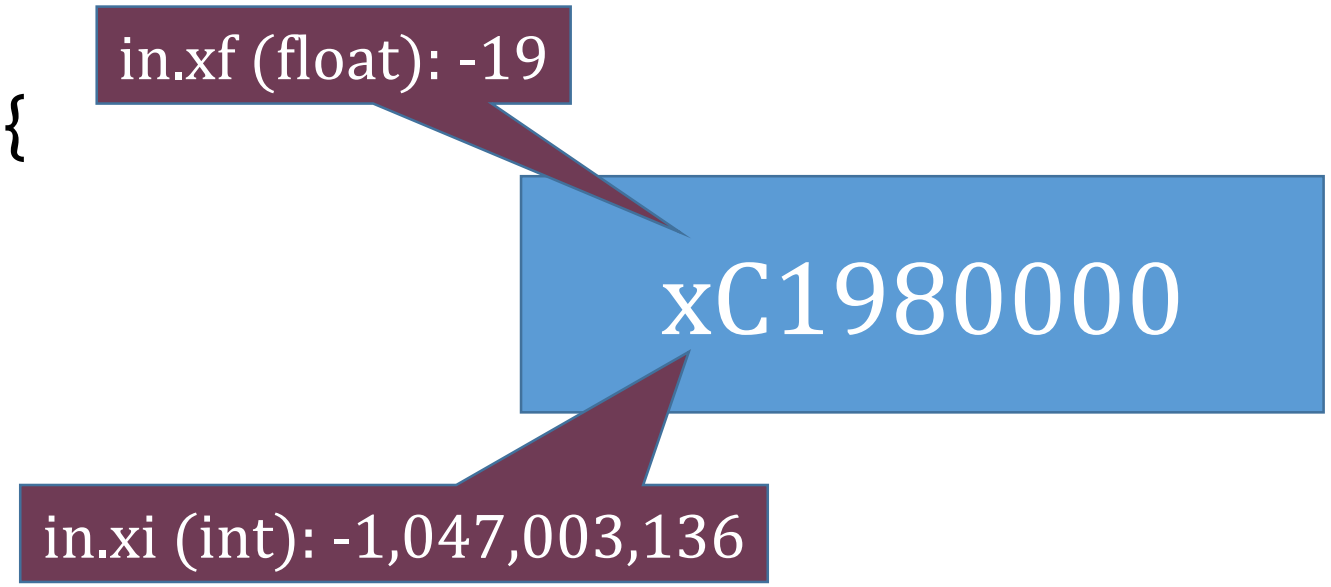


- Creates a new data type: “union flint”
- Does NOT create any variables with this data type!
- Does NOT reserve any memory

Example of Union: Test 1 “hexRep”

```
int hexRep(float x) {
        union {
                float xf;
                int xi;
        } in;
        in.xf=x; // Copy float parameter to union as float
        return in.xi; // Return value of union as an integer
}
```

Example of Union: Test 1 “hexRep”




```
int hexRep(float x) {
        union {
                float xf;
                int xi;
        } in;
        in.xf=x; // Copy float parameter to union as float
        return in.xi; // Return value of union as an integer
}
```

Example Union to show endian-ness

```
union {
        int i;
        unsigned char ic[4];
} iu;
iu.i=12;
printf("Integer %d is %02x %02x %02x %02x\n",
        iu.i, iu.ic[0], iu.ic[1], iu.ic[2], iu.ic[3]);
/* Integer 12 is 0c 00 00 00 */
```

Label	Address	Value
iu.ic[3]	0xc003	0x00
iu.ic[2]	0xc002	0x00
iu.ic[1]	0xc001	0x00
iu.i, iu.ic[0]	0xc000	0x0c

What is an Enumeration (enum)?

- A data type characterized by a finite set of named elements
- A mapping from names to integers

enum [<tag>] { <memlist> } [<instname>];

enum example

```
enum scr_colors { red, green, blue, gray } fg_color=blue;
enum scr_colors bg_color;
bg_color=gray;

if (fg_color==bg_color) printf("Who turned off the lights?\n");
```

enum under the covers

```
enum scr_colors { red, green, blue, gray } fg_color=blue;
```

- fg_color is a number
- The type of fg_color depends on how many different colors there are.. if there are less than 16 colors, fg_color is an unsigned char
- Compiler maps red->0, green->1, blue->2, gray->3
- Compiler limits values of type scr_colors to values : red, green, blue, or gray

C enum's - concrete

```
// enum scr_colors { red, green, blue, gray } fg_color;
/* after type checking same as…*/
unsigned char fg_color;
#define red 0
#define green 1
#define blue 2
#define gray 3
```

Why enum?

- Type Checking!
  - Compiler error if you say “fg_color=cyan;”
  - Compiler error if you say “scr_color bg_color = fg_color * 3;”
- Makes code much more legible
  - if (color==red) vs if (color==2)

What is a typedef?

- A new name for a type which is already defined

typedef <old_type> <new_type>

Example typedef

```
#include <stdbool.h>
typedef unsigned char[3] area_code;
area_code local={6,0,7};

bool ac_equal(area_code ac1, area_code ac2) {
        if (ac1[0]!=ac2[0]) return false;
        if (ac1[1]!=ac2[1]) return false;
        if (ac1[2]!=ac2[2]) return false;
        return true;
}
```

Combining typedef with structures

```
typedef struct { int x; int y; int z; } loc3d;

loc3d origin={0,0,0};
loc3d pos=origin;
pos.x=pos.x+3; // move 3 spaces forward
```

"new type"

"existing type" : anonymous structure definition