# struct

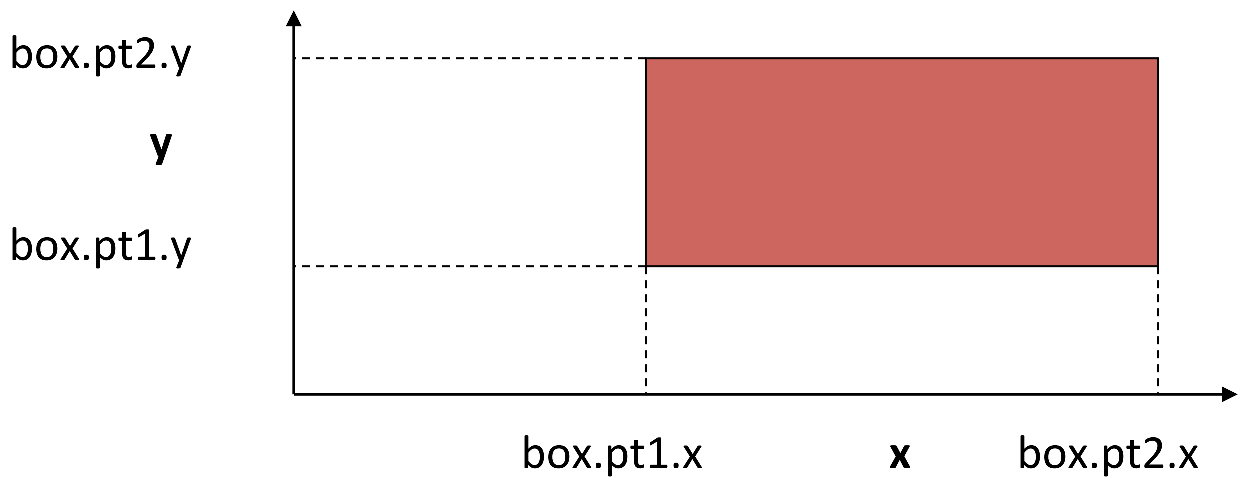
- A **struct** is a collection of variables, possibly of different types, grouped under a single name for common reference as a unit.

```
struct point {      /* with optional tag */
    int x;          /* member x          */
    int y;          /* member y          */
} ;
struct point q= {320, 200};
struct point pt;
pt.x = 320;  pt.y = 200;
```

# struct

- A struct inside a struct (nesting).

```
struct rect {
        struct point pt1; /* lower left      */
        struct point pt2; /* upper right     */
 };
 struct rect box;/* declare box as a rect */
```

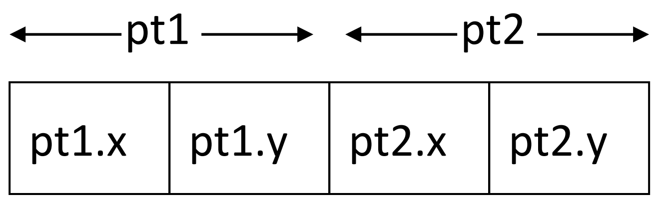


# struct

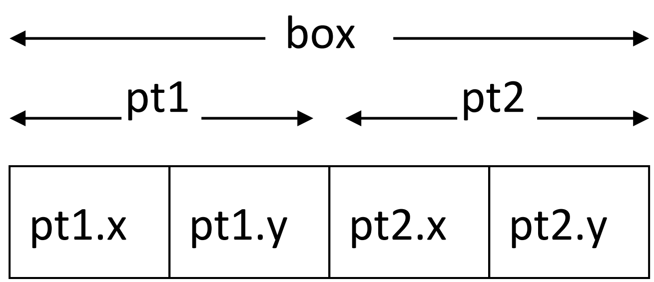
```
/* Find area of a rectangle */
int area = rectarea (box);
…
int rectarea (struct rect x) {
 return (x.pt2.x - x.pt1.x) * (x.pt2.y - x.pt1.y);
}
```

# struct

- Memory allocation for structs
  - Two point structs pt1 and pt2




  - One rect struct box containing two point structs



# What can we do with a struct?

- Reference members

  **box.pt2.x = box.pt1.x + width;**

- Assign as a unit

  **pt2 = pt1;**

- Create a pointer to it

  **struct point *ppt1;**

  **ppt1 = &pt1;**

# What can we do with a struct?

- Not legal to compare structs

**if (pt1 == pt2)** …  ← INVALID

- Must be done as:

**if (pt1.x == pt2.x && pt1.y == pt2.y)** …

# struct and Functions

We can use a struct as any other type with functions

```
/* check if 2 rectangles overlap */
int ptinrect (struct point p, struct rect r)
{
   return p.x >= r.pt1.x && p.x <= r.pt2.x
    && p.y >= r.pt1.y && p.y <= r.pt2.y;
}
```

# Arrays of structs

- Multiple related arrays to store a number of keywords and their corresponding counts

```
char * keyword[NKEYS];
int keycount[NKEYS];
```

- Can be implemented as an array of structs

```
struct key {
   char *word;
   int count;
};
struct key keytab [NKEYS];
```

# Arrays of structs

- Alternative array of structs implementation

```
struct key {
   char *word;
   int count;
} keytab[NKEYS];
```

# Arrays of structs

- **Initialization** for an array of structs

```
struct key {
    char *word;
    int count;
} keytab[ ] = {
    "auto", 0,
     …
     "while", 0
};      /* NKEYS is dynamically derived */
```

# Pointers to structs

- Declare and initialize a pointer to struct

  **struct point p;**

  **struct point *pp = &p;**

- Refer to members of struct p via pointer pp

  **(*pp).x** and **(*pp).y**

  or

  **pp->x** and **pp->y**

# Pointers to structs

```
struct string {
   int len;
   char *cp;
} *p;
```

| Expression | Same as | Value / Effect |
| --- | --- | --- |
| ++p->len | ++(p->len) | incr len |
| *p->cp | *(p->cp) | value is a char |
| *p->cp++ | *((p->cp)++) | value is a char incr cp |

# Pointers to structs

```
/* check if two rectangles overlap
   ptinrect:    (pointer version)
   if point p in rect r, return 1 else return 0
*/

int ptinrect (struct point *pp, struct rect *rp)
{
   return pp->x >= rp->pt1.x && pp->x <= rp->pt2.x
    && pp->y >= rp->pt1.y && pp->y <= rp->pt2.y;
}
```

# Review

- Declarations/Definitions, K&R pg. 9, 210
- A <u>declaration</u> specifies the interpretation to be given to an identified variable
- A <u>definition</u> is a declaration that reserves storage
- In C89, declarations/definitions must be made before <u>all</u> executable statements in the same block {…}
- NOTE: In C99, this requirement is relaxed. All declarations/definitions must be made before being used in any executable statements.

# Compiler vs Dynamic Allocation of Memory

- If size is known at "compile time" (i.e., it is a pre-defined constant value):
  - Use the compiler to allocate memory
  - This is the easiest way
- If size is NOT known at "compile time"(e.g., it is entered by user at run time):
  - You must use dynamic allocation of memory
  - Be careful to use **malloc**( ) and **free**( ) correctly

# Compiler Allocated Array

```
static char *lines[MAXLINE];      /* define array */
```

/* Static array elements are initialized = NULL for you */

```
. . .
```

/* No need to free memory allocated for the array lines */

/* (The compiler takes care of all the details for you) */

# Dynamically Allocated Array

- If you had no specification for maximum number of lines tail had to be able to hold
- If you only get maximum number (n) at run time:

```
static char **lines;   /* pointer to array of pointers */
lines = (char **) malloc (n * sizeof (char *));
```

/* You must initialize all pointers in array = NULL */

/* Later, you must free memory allocated for the lines array <u>after</u> freeing memory for each line */

```
free ( (void *) lines);
lines = NULL;
```

# Static/Dynamic Arrays

- Regardless of how `lines` array is declared – static or dynamic, refer to it in the same way:

```
lines[i] = (char *) malloc (strlen(line)+1);
strcpy (lines[i], line);
printf ("%s\n", lines[i++]);
free( (void *) lines[i]);
lines[i] = NULL;
```

# Compiler Allocated Array of structs

```
struct pet {
        char *type;
        char *name;
};                  /* each instance of struct pet is 8 bytes */
void print_list(struct pet *, int);        /* function prototype */

int main ()        /* array and structs allocated by compiler  */ {
                   /* all are automatic – allocated on stack */
    struct pet list [ ] = {{"cat", "fluffy"}, {"dog", "spot"}};
    print_list(list, sizeof list / sizeof(struct pet));  /* size = 2 */
    return 0;
}                  /* all memory used goes "poof" upon return */
```

# Dynamically Allocated Array of structs

```
int main () {
    int n = 2;
    /* all structs are in dynamic memory */
    struct pet *list = (struct pet *) malloc (n * sizeof(struct pet));
    list[0].type = “cat”;
    list[0].name = “fluffy”;
    list[1].type = “dog”;
    list[1].name = “spot”;
    print_list(list, n);
    free ( (void *) list);   /* free memory used for the array */
    list = NULL;   /* optional - goes “poof” on return */
    return 0;
}
```

# Dynamically Allocated Array of structs

```
int main () {
    int n = 2;
    /* all structs are in dynamic memory */
    struct pet *list = (struct pet *) malloc (n * sizeof(struct pet));
    struct pet *copy = list;
    copy->type = “cat”;     /* “real” C programmer’s way */
    copy->name = “fluffy”;
    (++copy)->type = “dog”;
    copy->name = “spot”;
    print_list(list, n);
    free ( (void *) list);   /* free memory used for the array */
    return 0;                /* list and copy go out of scope     */
}
```

# Print Either Array of structs

```
void print_list(struct pet *list, int size) {
    while (size--) {
     /* defensive programming – check pointer values */
     if (list != NULL && list->type != NULL && list->name != NULL)
          printf("%s, %s\n", list->type, list->name);
     list++;
   }
}
```

# Introduction to C99

- C99 standard relaxes a few of the C89 rules
  - // style single line comments may be used
  - Variables may be defined anywhere in a block. They do not need to be defined before all executable code (useful and helps to enable the next 2 rules)
  - Automatic array variables may be dimensioned with a value known at run time – not just with a constant
  - Loop variable may be declared in the initialization statement of a "for" loop– similar to Java
  - New complex and boolean data types are supported

# Introduction to C99

- gcc supports C99 with the –c99 option flag
- C89 code will compile correctly under C99
- You aren' t required to use any C99 features!
- You may be able to use C99 to simplify the code for your particular program
- Careful - if there is any requirement for your code to be C89 compatible for any reason!

# Dynamically-sized Arrays (C99)

- If you have no specification for maximum number of lines that the lines array has to be able to hold
- If you only get the value of n at run time:

```
char *lines [n];     /* cannot be "static" or "external" */
/* You must initialize all pointers in array = NULL */
/* Normal rules of "block" scope for compiler-allocated
   memory apply!  DON'T CALL FREE! */
```

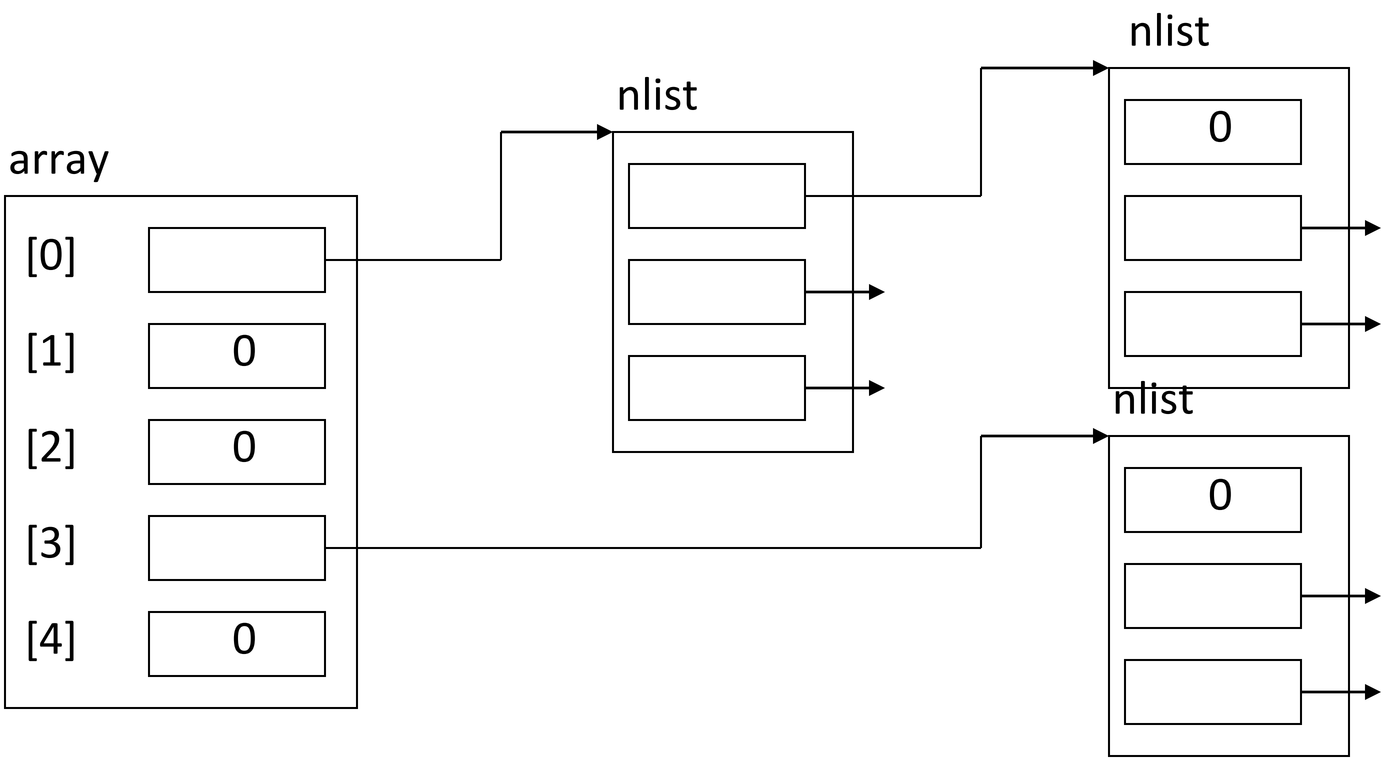
# Table Lookup, K&R 6.6

- struct for a linked list of names/definitions

```
struct nlist {                /* table entry      */
    struct nlist *next;       /* link to next     */
    char *name;               /* word name        */
    char *defn;               /* word definition */
};
```

# Table Lookup / Linked Lists

- Array of pointers to null terminated linked lists of structs defining table entries



# Table Lookup / Linked List

- Hash function to select starting array element

```
unsigned int hash (char *s)
{
for (hashval = 0; *s != '\0' ; s++)
    hashval = *s + 31 * hashval;
return hashval % HASHSIZE;
}
```

- In lookup ( ), std code for following a linked list:

```
for (ptr = head; ptr != NULL; ptr = ptr->next)
```

# Multi-Dimensional Arrays

- Declare rectangular multi-dimensional array

```
char a[2][3]; /* array [row] [column] */
```

- NOTE: `char a[2, 3]` is <u>INCORRECT</u>!
- The rightmost subscript varies fastest as one looks at how data lies in memory:

```
a[0][0], a[0][1], a[0][2], a[1][0], a[1][1], ...
```

- It is the same as a one dimensional array [2] with each element being an array [3]

# Multi-Dimensional Arrays

- Example of converting a month & day into a day of the year for either normal or leap years

```
static char daytab[2][13] = {
{0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31},
{0, 31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31}
};
```

- Use a second row of day counts for leap year rather than perform a calculation for days in February

  `daytab[1][1]` is 31 → same as `daytab[0][1]`

  `daytab[1][2]` is 29 → not same as `daytab[0][2]`

# Multi-Dimensional Arrays

- The array declared as `char daytab[2][13]` can be thought of as:

```
char (daytab [2]) [13];  /* pg. 53 */
```

- Each one dimensional array element (`daytab[0]`, `daytab[1]`) is like array name - as if we declared:

```
char daytab0 [13], daytab1 [13];
```

- `daytab[0]` is in memory first, then `daytab[1]`

# Multi-Dimensional Arrays

- `daytab[0]` and `daytab[1]` are arrays of 13 chars
- Now recall duality of pointers and arrays:

```
(daytab [0])[n]  →  (*daytab)[n]

(daytab [1])[n]  →  (*(daytab+1))[n]
```

- `daytab` is a pointer to an array of elements each of which is an array of size 13 chars

# Multi-Dimensional Arrays

- But, these two declarations are not allocated memory the same way:

  `char daytab[2][13];` → 26 char-sized locations

  `char (*dp)     [13];` → 1 pointer-sized location

- For the second declaration, code must set the pointer equal to an already defined array of [n] [13] `dp = daytab;`

  OR

- Use malloc to allocate memory for an array: `dp = (char (*)[13]) malloc(2*13);`

# Multi-Dimensional Arrays

```
static char daytab[2][13] = { ... };
```

```
daytab[0][2] == 28          daytab[1][2] == 29
```

| Array of 13 chars | Array of 13 chars |
|---|---|

```
(*dp)[2] == 28              (*(dp + 1))[2] == 29
```

dp

```
dp = daytab;
```

1 char (*) [13] ←----- A pointer to an unspecified number of 13 char arrays

```
dp = (char (*)[13]) malloc(2*13);
```

| Array of 13 chars | Array of 13 chars |
|---|---|

```
(*dp)[2] == ??              (*(dp + 1))[2] == ??
```

# Multi-Dimensional Arrays

- “Real” C programmers use pointers to pointers often and multidimensional arrays rarely
- Avoids worst case for memory allocation

# typedef, K&R 6.7

- typedef creates a new name for existing type

```
typedef int Boolean;
typedef char *String;
```

- Does not create a new type in any real sense
- No new semantics for the new type name!
- Variables declared using new type name have same properties as variables of original type

# typedef

- Could have used typedef in section 6.5 like this:

```
typedef struct tnode {
 char *word;
 int count;
 treeptr left;
 treeptr right;
} treenode;
typedef struct tnode *treeptr;
```

# typedef

- Then, could have coded talloc ( ) as follows:

```
treeptr talloc(void)
{
 return (treeptr) malloc(sizeof(treenode));
}
```

# typedef

- Used to provide clearer documentation:

```
treeptr root;
```

versus

```
struct tnode *root;
```

- Used to create machine independent variable types:

```
typedef int size_t;/* size of types */
typedef int ptrdiff_t; /* difference of pointers */
```

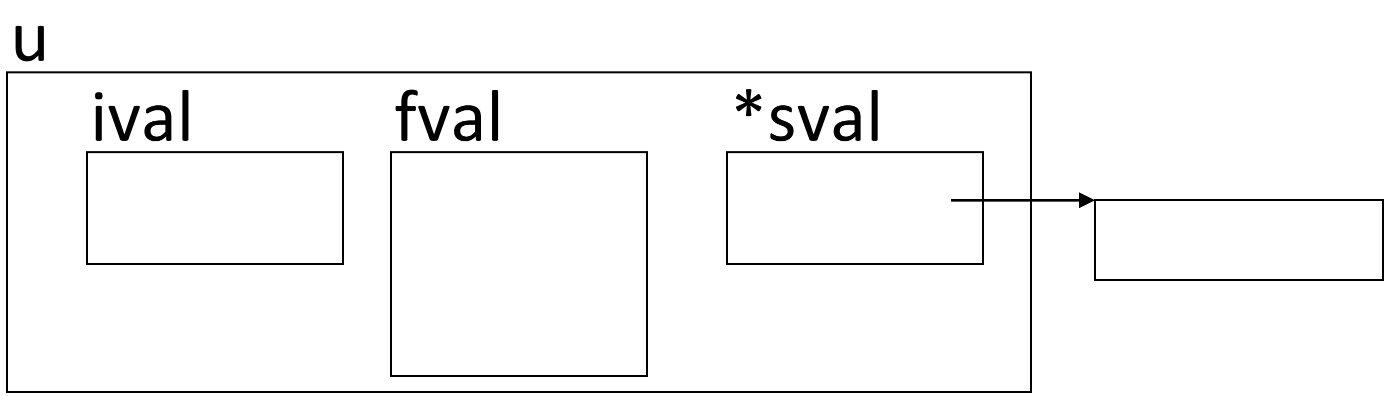
# Unions, K&R 6.8

- A Union is a variable that may hold objects of different types (at different times or for different instances of use)

```
union u_tag {
    int ival;
    float fval;
    char *sval;
} u;
```

# Unions

- A union will be allocated enough space for the largest type in the list of possible types
- Same as a struct except all members have a zero offset from the base address of union



# Unions

- The operations allowed on unions are the same as operations allowed on structs:
  - Access a member of a union

```
union u_tag x;
x.ival = … ;
```

  - Assign to union of the same type

```
union u_tag y;
x = y;
```

# Unions

- Create a pointer to / take the address of a union

```
union u_tag x;
union u_tag *px = &x;
```

- Access a member of a union via a pointer

```
px->ival = … ;
```

# Unions

- Program code must know which type of value has been stored in a union variable and process using correct member type
- DON'T store data in a union using one type and read it back via another type in attempt to get the value converted between types

```
x.ival = 12; /* put an int in union */
float z = x.fval;  /* don't read as float! */
```

# Bit-Fields, K&R 6.9

- Bit fields are used to get a field size other than 8 bits (char), 16 bits (short on some machines) or 32 bits (long on most machines)
- Allows us to "pack" data one or more bits at a time into a larger data item in memory to save space, e.g. 32 single bit fields to an int
- Can use bit fields instead of using masks, shifts, and or's to manipulate groups of bits

# Bit-Fields

- Uses struct form:

```
struct  {
   unsigned int flg1 : 1; /* called a “bit field“          */
   unsigned int flg2 : 1; /* ": 1" →"1 bit in length " */
   unsigned int flg3 : 2; /* ": 2" →"2 bits in length" */
} flag;    /* variable */
```

- Field lengths must be an integral number of bits

# Bit-Fields

- Access bit fields by member name same as any struct – just limited number of values

```
flag.flg1 = 0;  /* or = 1;  */
flag.flg3 = 0;  /* or = 1; = 2; =3;  */
```

# Bit-Fields

- Minimum size data item is implementation dependent, as is order of fields in memory
- Don't make any assumptions about order!!
- Sample memory allocation: