Lecture 13. Structures

- **An array is a *homogeneous* collection: all of its elements have the *same type***
- **A structure is a *heterogeneous collection*: its elements can have *different* types**

```
struct date {
    int day;
    int month;
    int year;
    char monthname[4];    /* "Jan", "Feb", etc. */
};
```

Declares a *new type*, `struct date`, with four named elements, called *fields*

- **Structures can be *nested***

```
struct student {
    char name[30];
    float gpa;
    struct date birthday;
};
```

- **Structure types can be used like `int`, `float`, etc. to declare variables and arrays, which can optionally be initialized — and they must be initialized before use**

```
struct date today;
struct student cs126[140];

struct date bday = { 2, 11, 1977, "Nov" };
```

Fields

- **Structure fields are accessed by *variable*`.`*field***

`bday.day`	the day field in bday, the `int` 2
`bday.name[i]`	the `i`th character in the `monthname` field of bday, a `char`

- ***Field selection*** **operator associates to the *left* and has high precedence**

```
struct student cs126[140];
```

`cs126[i].gpa`	the GPA of the `i`th `student` in `cs126`
`cs126[i].name[j]`	the `j`th character in the name of the `i`th `student`
`cs126[i].birthday.year`	the year of the `i`th `student`'s `birthday`
`cs126[i].birthday.monthname[0]`	the first letter in the `monthname` of the `i`th `student`'s `birthday`

- **Field selection denotes an *lvalue*; use assignments to initialize/change field values**

```
today.day = 24;
today.month = 10;
today.year = 1996;
strcpy(today.monthname, "Oct");

swap(&today.day, &bday.day);
```

Arrays of Structures

- **A structure type provides a way to package related data in one variable**

```
struct card {
    char *face;
    char *suit;
};

char *suits[] = { "Hearts", "Diamonds", "Clubs", "Spades" };

char *faces[] = { Ace", "2", "3", "4", "5", "6", "7", "8",
    "9", "10", "Jack", "Queen", "King" };

int main(void) {
    int i;
    struct card deck[52];

    deck[0].face = faces[0]; deck[0].suit = suits[0];
    deck[1].face = faces[1]; deck[1].suit = suits[0];
    for (i = 2; i < 52; i++) {
        int k = rand()%i;
        deck[i] = deck[k];
        deck[k].face = faces[i%13]; deck[k].suit = suits[i/13];
    }
    for (i = 0; i < 52; i++)
        printf("%s of %s\n", deck[i].face, deck[i].suit);
    return 0;
}
```

Once shuffled, cards are represented by `struct card` values, not integers 0..51

Pointers to Structures

- **A *structure pointer* holds the address of a structure variable**

```
struct date today, bday, *pdate;

pdate = &today;              assigns the address of today to pdate
(*pdate).day = 2;            sets the day field of today to 2
(*pdate).year++;             increments the year field of today
printf("%s %d, %d\n", (*pdate).monthname, (*pdate).day,
    (*pdate).year);          prints the date given by today
bday = *pdate;               assigns today to bday, field-by-field
```

- **Structure pointers can ‘walk along’ arrays of structures**

```
struct card *dptr;

dptr = deck;
for (i = 0; i < 52; i++) {
    printf("%s of %s\n", (*dptr).face, (*dptr).suit);
    dptr++;
}

dptr = dptr + 1;    increment dptr means
dptr +=1;                ‘advance dptr to the next struct card element’
dptr++;                  not ‘add 1 to dptr’
```

Pointers to Structures, cont’d

- (*ptr).*field* is so common that there’s an abbreviation: *ptr*->*field*

use	*var*.*field*	when *var* is a ***structure***
use	*var*->*field*	when var is a ***pointer to a structure***
or	(**var*).*field*	

`->` has high precedence, but less than `.`

`pdate->day = 2;`	sets the day field of `*pdate` to 2
`pdate->year++;`	increments the year field of `*pdate`

`printf("%s %d, %d\n", pdate->monthname, pdate->day,`
`    pdate->year);`
	prints the date given by `*pdate`

```
for (i = 0; i < 52; i++) {
    printf("%s of %s\n", dptr->face, dptr->suit);
    dptr++;
}
```

- Pointer madness! Structures can contain other pointers, but watch precedence

```
struct foo { int x, *y; } *p;
```

`++p->x`	increments field x in `*p`
`(++p)->x`	increments p, ***then*** acesses field x
`*p->y++`	returns the `int` pointed to by field y in `*p`, increments y
`*p++->y`	returns the `int` pointed to by field y in `*p`, increments p

Typedefs

- ‘`struct card`’ is a bit wordy and can make code hard to read
- A `typedef` ***associates*** an ***identifier*** with a ***type***, which makes code more readable

```
typedef struct card Card;
```

Declares `Card` to be a type name for ‘`struct card`’
`Card` may be used anywhere `struct card` can be used

Case matters!

Putting it all Together: Card Shuffling Revisited

- Represent a deck by an ***array of pointers to cards***; shuffle by rearranging the pointers, not the cards themselves

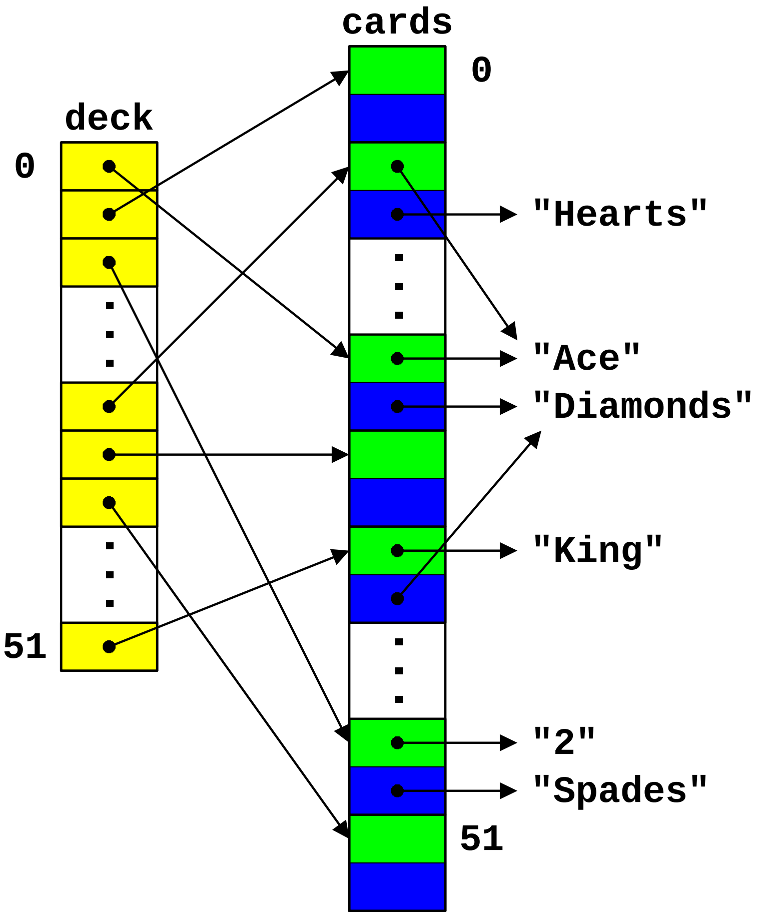
```
typedef struct card Card;

struct card {
    char *face;
    char *suit;
};

Card cards[52];

void shuffle(Card *deck[52]) {
    int i;

    deck[0] = &cards[0];
    deck[1] = &cards[1];
    for (i = 2; i < 52; i++) {
        int k = rand()%i;
        deck[i] = deck[k];
        deck[k] = &cards[i];
    }
}
```



Card Shuffling Revisited, cont’d

- **Mapping of 0..51 onto faces and suits is confined to initialization**

```
char *suits[] = { "Hearts", "Diamonds", "Clubs", "Spades" };

char *faces[] = { "Ace", "2", "3", "4", "5", "6", "7", "8",
    "9", "10", "Jack", "Queen", "King" };

void initialize(void) {
    int i;

    for (i = 0; i < 52; i++) {
        cards[i].face = faces[i%13];
        cards[i].suit = suits[i/13];
    }
}

int main(void) {
    int i;
    Card *deck[52];

    initialize();
    shuffle(deck);
    for (i = 0; i < 52; i++)
        printf("%s of %s\n", deck[i]->face, deck[i]->suit);
    return 0;
}
```

- **Can handle *many* decks (arrays of pointers) with only *one* array of card structures**