What have we learned?

struct

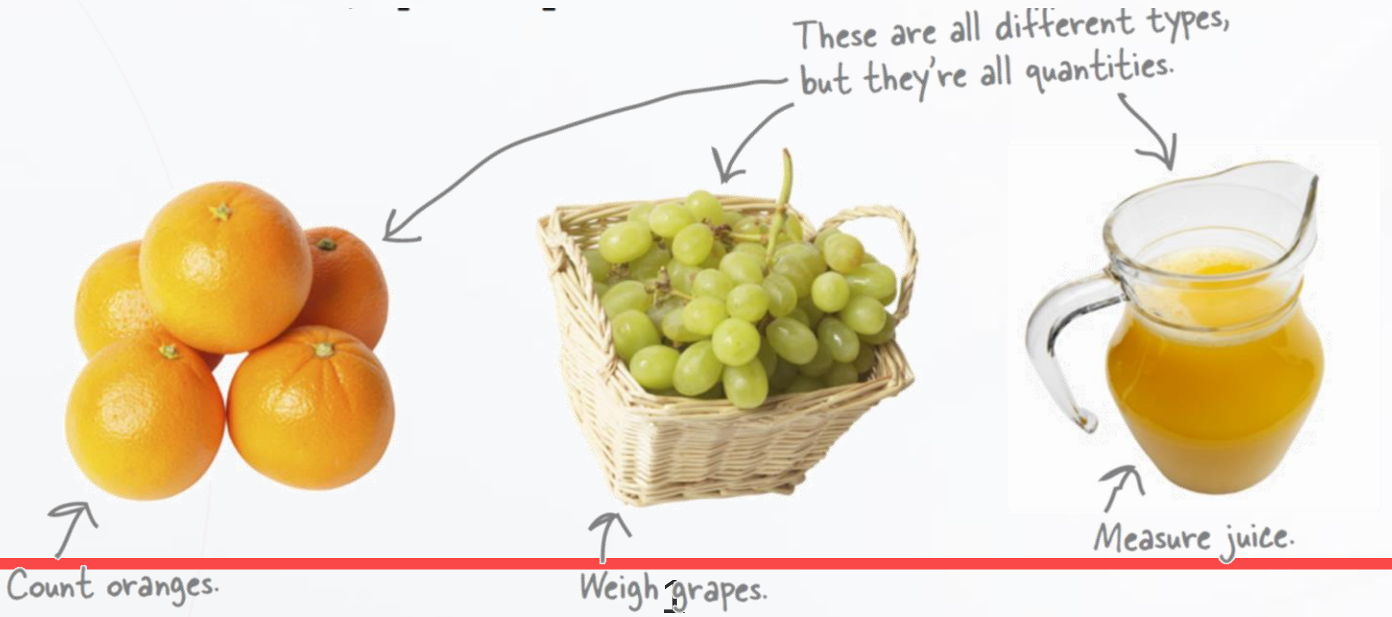
short count	float weight	float volume

```
typedef struct {
      short count;
      float weight;
      float volume;
} fruit;
```

union

quantity (short or float)

```
typedef union {
      short count;
      float weight;
      float volume;
} quantity;
```



Unions used together with structs

```
typedef union {
    float lemon;
    int lime_pieces;
} lemon_lime;

typedef struct {
    float tequila;
    float cointreau;
    lemon_lime citrus;
} margarita;
```

```
margarita m = {2.0, 1.0, {0.5}};
```

```
margarita m = {2.0, 1.0, {.lime_pieces=1}};
```

```
margarita m = {2.0, 1.0, .citrus.lemon=2};
```

Initializers are for initialization, not for assignments

```
margarita m = {2.0, 1.0, {0.5}};
```

✓

```
margarita m;
m = {2.0, 1.0, {0.5}};
```

✗

The complier regards this as an array!

An enum variable stores a symbol

- Sometimes you want to store something from a *list of symbols*, e.g. a day of the week, MON, TUE, WED, …
- **enum** let’s you create a list of symbols.

```
enum colors {RED, GREEN, PUCE};
```

You may also use typedef to give it an alias

```
enum colors favorite = PUCE;
```

```
enum colors favorite = PUSE;
```

✗

Anything not in the list will be rejected by the compiler

Example: use enum to keep track of what’s in union

```
typedef enum {
    COUNT, POUNDS, PINTS
} unit_of_measure;

typedef union {
    short count;
    float weight;
    float volume;
} quantity;

typedef struct {
    const char *name;
    const char *country;
    quantity amount;
    unit_of_measure units;
} fruit_order;
```

Example (continued)

```
int main()
{
    fruit_order apples = {"apples", "England",
                           .amount.count=144, COUNT};
    fruit_order strawberries = {"strawberries", "Spain",
                           .amount.weight=17.6, POUNDS};
    fruit_order oj = {"orange juice", "U.S.A.",
                           .amount.volume=10.5, PINTS};
    display(apples);
    display(strawberries);
    display(oj);
    return 0;
}
```

Example (continued)

```
void display(fruit_order order)
{
    printf("This order contains ");
    if (order.uni_of_measure == PINTS)
        printf("%2.2f pints of %s\n", order.amount.volume,
               order.name);
    else if (order.uni_of_measure == POINTS)
        printf("%2.2f lbs of %s\n", order.amount.weight,
               order.name);
    else
        printf("%i %s\n", order.amount.count,
               order.name);
}
```

```c
typedef enum {SUN, MON, TUE, WED, THU, FRI, SAT} DAY;
int main()
{
    DAY day_of_the_week;
    ... /* obtain day of the week */
    switch day_of_the_week
    {
    case SUN:
        ...; break;
    case MON:
        ...; break;
    case TUE:
        ...; break;
    case WED:
        ...; break;
    ...
    }
}
```

Lecture 8 Input & output

Input/output libraries

- `<stdio.h>` header is the primary repository of input/output functions, e.g. `printf`, `scanf`, `putchar`, `getchar`, `puts`, `gets`...

  Byte input/output functions

- `<wchar.h>` functions deal with wide characters rather than ordinary characters.

  Wide-character input/output functions

Streams

- In C, ***stream*** means any source of input or any destination for output.
- A small program obtains all input from one stream (keyboard) and writes all output to another (screen).
- Larger programs may need additional streams.
- Streams often represent files stored on various media.
- However, they could also be associated with devices such as network ports and printers.

Standard streams

File Pointer	Stream	Default Meaning
`stdin`	Standard input	Keyboard
`stdout`	Standard output	Screen
`stderr`	Standard error	Screen

Declared in `<stdio.h>`. No need to open or close.

- Standard streams may be redirected
  - Input redirection (in command line)

```
demo <in.dat
```

Obtains input from file “in.dat” rather than the keyboard.

  - Output redirection

```
demo >out.dat
```

Writes output to file “out.dat” rather than the screen.

```
demo <in.dat >out.dat
```

Standard stream redirection

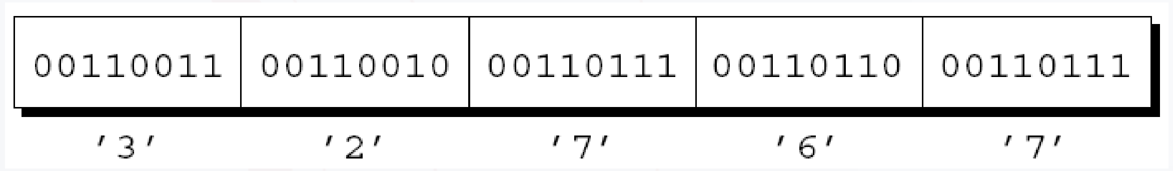
- Simplicity is one of the attractions of input and output redirection.
- Limitations of redirection
  - When a program relies on redirection, it has no control over its files; it doesn't even know their names.
  - Redirection doesn't help if the program needs to read from two files or write to two files at the same time.
- When redirection isn't enough, use the file operations in `<stdio.h>`

Text Files versus Binary Files

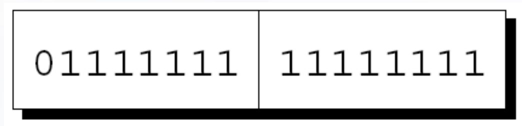
- A ***text file*** stores characters, allowing humans to examine or edit the file.
  - E.g. the source code for a C program.
- A ***binary file*** stores general data, which may *not* represent characters.
  - E.g. a executable C program.

Example: number 32767

Text



Binary



A file redirected from the standard I/O stream is usually a text file.

Opening a file

- A file needs to be opened before reading/writing

```
FILE *fopen(const char * filename,
            const char * mode);
```

- `Filename`: name of the file to be opened.
  - May include information about the file's location, such as a drive specifier or path.
- `mode` is a "mode string" that specifies what operations we intend to perform on the file.

Opening a file

- `fopen` returns a FILE pointer: (null pointer if fails)

```
fp = fopen("in.dat", "r");
   /* opens in.dat for reading */
```

- The call `fopen("c:\project\test1.dat", "r")` will fail, because `\t` is treated as a character escape.
- Use `\\` instead of `\`:

```
fopen("c:\\project\\test1.dat", "r")
```

- An alternative is to use the `/` character instead of `\`:

```
fopen("c:/project/test1.dat", "r")
```

Modes

	String	Meaning
Text files	"r"	For reading only
	"w"	For writing only (file may not exist)
	"a"	For appending only (file should exist)
	"r+"	For reading & writing (starting at beginning)
	"w+"	For reading & writing (overwritten if file exists)
	"a+"	For reading & writing (append if file exists)
Binary files	"rb"	For reading only
	"wb"	For writing only (file may not exist)
	"ab"	For appending only (file should exist)
	"rb+"	For reading & writing (starting at beginning)
	"wb+"	For reading & writing (overwritten if file exists)
	"ab+"	For reading & writing (append if file exists)

Closing a file

- The `fclose` function closes a file that is no longer in use:

```
int fclose(FILE *fp);
```

A file pointer obtained from `fopen` or `freopen`

- Returns `0` if the file closed successfully.
- Otherwise, it returns the error code `EOF` (a macro defined in `<stdio.h>`).

Formatted I/O

- `printf` and related functions convert data from *binary* form to *text* form during output.

variable number of arguments

```
int fprintf(FILE *stream,
             const char *format, ...);
```

return the number of characters written (negative for errors)

- `scanf` and related functions convert data from *text* form to *binary* form during input.

```
int fscanf(FILE *stream,
            const char *format, ...);
```

returns the number of input items successfully assigned