Struct

DECLARING A STRUCT

```
struct matrix {
    double **array;
    long num_of_rows;
    long num_of_cols;
};
```

INITIALISING A STRUCT VARIABLE

```
struct module cs1010
```

`struct module` — type; `cs1010` — var name

← "instance" of a struct

```
.code = "CS1010";
.title = "Programming Methodology";
.mc = 4;
```

. notation is used to access members/attributes of a certain struct

PASSING STRUCTS INTO A FUNCTION

As a pass-by value

```
void fn(struct type var_name, type other var){
}
```

As pass-by reference

```
void update_mc(struct module *cs1010, long num_hours){
    (*cs1010).mc = num_hours/2.5;
}
```

`*cs1010` ← pointer to struct

`(*cs1010).mc` ← dereferencing struct

We can dereference pointers to struct using the arrow notation:

```
CS101->mc = num_hours/2.5;
```

A struct gets copied when returning value to the caller

DEFINING OUR OWN TYPE in C

Convention dictates suffixing our own types using "_t".

This can be declared using typedef:

```
typedef struct module{
    char *code;
    char *title;
    long mc;
}

    OR

typedef struct{
    char *code;
    char *title;
    long mc;
} module;
```

REASONS FOR NOT USING TYPEDEF

- Makes code harder to read (used to refer to info obscured by creating our own types)

STDIO (Standard Input & Output)

printf

Example: `printf("%s is a %ld-MC module\n", module.code, module.mc)`

- `%s` — string
- `%ld` — long
- `\n` — newline

printf can take in variable number of parameters

Format: `%[flags][field_width][.precision][length_modifier]specifier`

scanf

```
long width;
long height;
scanf("%ld %ld", &width, &height)
```

`&width` — memory address of width

Scanning stops when an input character does not match such a format character or when input conversion fails.

On success scanf returns number of items successfully matched. Useful for debugging.