C Lecture 4

- Reading Strings: fgets(), scanf(), and sscanf() functions
- Structured data: structs
- Functions in C

fgets(); scanf(); sscanf()

```
char[30] name;
int number = 0;
printf("Enter name and phone number: ");
fgets… ?
```

`fgets()` reads the whole line as a single string. What do you do if you want to read a line in that contains a number or more than one thing (e.g. two strings; or a mixture of ints, floats, and strings)?

Two options:

1.Use the `scanf()` function instead, which is analogous to `printf()`, and allows you to specify one or more format conversions.

2.Continue to use `fgets()` to read in the whole line but then use `sscanf()` to re-read the string (internally) and break it up into parts.

The `scanf()` Function

- The direct equivalent to the output function `printf` is the input function `scanf`. The syntax of a `scanf` statement is `scanf(format, &variable1, &variable2, . . . )` where `format` specifies the types of variables and `&variable1` is the address of variable1.
- A typical `scanf` statement would be `scanf("%d%d%f", &a, &b, &x)` reading in integer values for the variables a and b and a floating point value for the variable x, entered from the keyboard.
- The `%s` conversion in `scanf` ignores leading blanks and reads until either the end-of-string '\0' or the first blank after non-blank characters. For example, if the input from the keyboard is " ABCD EFG ", `%s` will read "ABCD".

Reading Numbers with `fgets()` and `sscanf()`

- Historically, scanf() was a bit unreliable at handling end-of-lines in some implementations (but now it's generally fine) so the combination of fgets() and sscanf() was often used.

Read in a number from the keyboard and double it

```
#include <stdio.h>
char  line[100];   /* input line from console */
int   value;       /* a value to double */

int main()
{
    printf("Enter a value: ");

    fgets(line, sizeof(line), stdin);
    sscanf(line, "%d", &value);

    printf("Twice %d is %d\n", value, value * 2);
    return (0);
}
```

} Equivalent to: scanf("%d" , &value);

Summary - Example

```c
#include <stdio.h>
#include <string.h>

char line[50];
char name[20];
int number;

int main() {
    puts("Enter name and number: ");
    fgets(line, sizeof(line), stdin);

    sscanf(line, "%s %d",name, &number);
    printf("\n Name and number are %s, %d \n\n",name,number);

    puts("Enter name and number: ");
    scanf("%s %d", name, &number);
    printf("\n Name and number are %s, %d \n\n",name,number);

    return 0;
  }
```

0

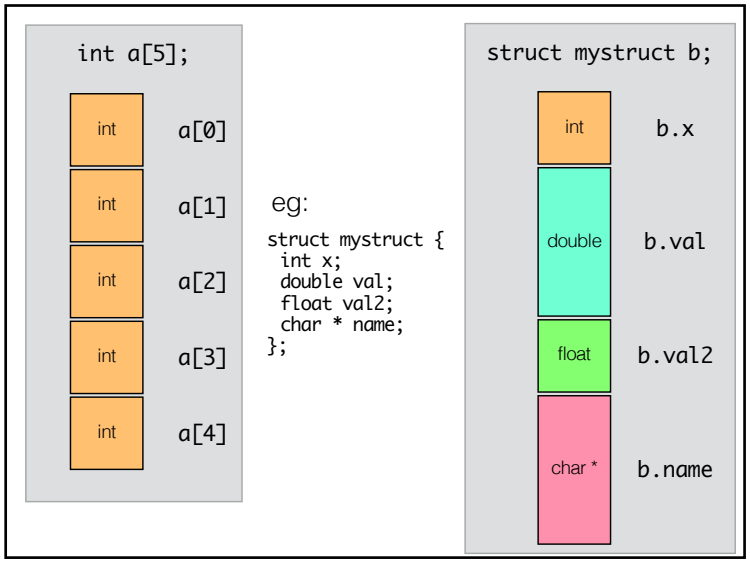
Structured Data

- Last lecture, we saw how array types let us group multiple values of the *same* type.
- Often, though, we have several pieces of data that make sense to keep together, but are of different types.
- For example:
  - name, account number, address in billing system;
  - particle x,y,z coords, vx,vy,vz coords, "particle type" (as a char) in a physics simulation.

Structured data types

* structs (short for *structured types*) provide this functionality in C.
* We can declare a name for a struct type for our particle like so.
* We can then use that name to make variables with the requested internal structure, as long as we prepend struct to let the compiler know the context.

```c
struct particle {
  double x,y,z;
  double vx,vy,vz;
  char ptype;
};
struct particle electron;
// an array of particles
struct particle p_array[5];
```



typedefs and structs

* The `typedef` keyword can be used to help "simplify" declaring structured types.
* `typedef` lets us specify a synonym for an existing type (including a struct type).
* If we use it for a `struct`, we can "typedef away" the need for the leading `struct` keyword when making variables of that type in future.
* Here, we tell C that when we say "`particle_t`", we mean to say "`struct particle`".

```
//convention: end new typenames with a _t
typedef struct particle particle_t ;
particle_t electron;
```

Initialising Structs

* You can initialise a struct type variable using the { } initialiser format you used for arrays.
* If we just list values, then they are assigned to the members of the struct in the order the members are defined (remaining members get set to 0).
* You can also explicitly mention a member name, prepended with a . to assign a value to.

```
struct particle {
  double x,y,z;
  double vx,vy,vz;
  char ptype;
};
```

```
// p.x=3, p.y=4, p.z=5
// p.ptype = 'e'
struct particle p = {3, 4, 5, .ptype='e' };
```

Equiverlent to:

```
struct particle p = {3, 4, 5, 0, 0, 0, 'e'};
```

Accessing components

We can access elements of a structured type by attaching the element name to the variable, with a joining .

```
p.x = 3.0;

int a = p.y *2;
switch(p.ptype) {
  case 'e':
    puts("This is an electron.");
    break;
  //more code here
  default:
    puts("Unknown particle type.");
  }
```

Accessing components

We can access elements of a structured type by attaching the element name to the variable, with a joining .

```
p.x = 3.0;

int a = p.y *2;
switch(p.ptype) {
  case 'e':
    puts("This is an electron.");
    break;
  //more code here
  default:
    puts("Unknown particle type.");
  }
```

Structs example

```c
#include <stdio.h>

struct particle {
        double x,y,z;
        double vx,vy,vz;
        char ptype;
    };

typedef struct particle particle_t;

int main() {


// create and initalise one particle using standard method
    struct particle p = {1,2,3,4,5,6,'e'};

// create and partially initialise another particle using typedef method
    particle_t q = {11,22,33, .ptype='n'};
    q.vx = 34.345;
    q.vy = 36.123;
    q.vz = q.vx * 2.2;

    printf("\n x = %f vx = %f type = %c\n\n",p.x, p.vx, p.ptype);
    printf("\n x = %f vx = %f type = %c\n\n",q.x, q.vx, q.ptype);
    return 0;
    }
```

Functions

- Often you will write a piece of code which solves a common problem.
- While *loops* let you repeat a block of code multiple times, you may want to use the same "solution" in different parts of your code, without having to rewrite it each time.
- *Functions* provide a way of "encapsulating" a chunk of code, and giving it a name so you can use it at multiple places.
- (You've already met several standard functions: `printf`, `sscanf`, `fgets` and so on.)

Function declarations

* Before we can use functions, we need to be able to declare them.
* A function declaration has two parts:
* The first part, the *function prototype*, declares the *type signature* (and type) of the function, along with its name.
* The second part, the *function body*, is a block which contains the statements that we want executed each time we call the function.

```c
int f(int a);

int main(void) {
   float x = 1.0;
   return f(x);
}

int f(int a) {
   a = a + 1;
   return a;
}
```

Forward declarations

* Just as a variable declaration doesn't have to assign a value, a function prototype does not have to be followed by a function body.
* However, a function prototype for a given function does have to occur in the *file* scope, *before* the function is used in any code in the file.
* A later declaration of the function body (complete with matching prototype) must be provided.
* The "early" function prototype is called a *forward declaration*.

```c
int f(int a);

int main(void) {
   float x = 1.0;
   return f(x);
}

int f(int a) {
   a = a + 1;
   return a;
}
```

Function Definitions

* The function body is a block, with the usual scoping rules for variables declared in it.
* That is: only file scope variables, and variables defined in the body itself, are in scope.
* The parameters of a function count as variables declared in the block scope of the body.
* The values of any parameters in the function call are *copied* to the variable names in the function scope, before the rest of the function runs.

```
int f(int a);

int main(void) {
   float x = 1.0;
   return f(x);
}

int f(int a) {
   a = a + 1;
   return a;
}
```

Parameters, Return types

* In one sense, a function is like a variable with a value (the return type) that is calculated each time it is used (from some inputs).
* Compare with a mathematical function, which maps a domain to a range.

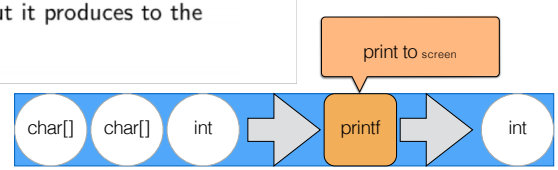
```
int f(int a);

int main(void) {
   float x = 1.0;
   return f(x);
}

int f(int a) {
   a = a + 1;
   return a;
}
```

Side Effects

* For some functions, the most important aspect of their existence is the things they do *other* than returning a result; their *side effects*.
* `printf`, for example, is such a function: while it returns a value (the number of characters it printed), the most important thing is the output it produces to the screen.



Functions example

```
#include <stdio.h>

// function prototype here
int f(int a); // int f(int) would be sufficient for prototype

int main(void) {
    int b = 0;
    char buffer[10];

    puts("Enter an Integer value:");
    fgets(buffer,sizeof(buffer),stdin);
    sscanf(buffer,"%d",&b);

    printf("You typed: %d \n\n",b);

    /* we can use f here, even though we've defined it later on
      as the prototype is above */
    printf("Result is: %d \n\n",f(b));

    //C passes by value, so b itself is unchanged
    printf("Value in b is still: %d\n\n",b);

    return 0;
}

//Function body here
int f(int a) { //we do need the a here - value of b is copied to a
    a += 1;
    //or a++; or a = a+1;

    return a; //this value is then the "result" of f
}
```