C Lecture 4

"structured data", Functions, and pointers

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.

```
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' };
```

Accessing components

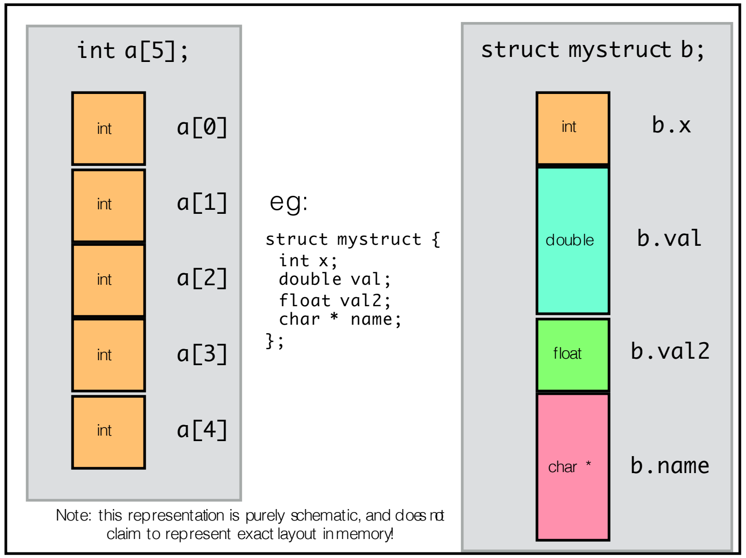
We can access elements of a structured type by attaching the element name to the variable, with a joining . .

```
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

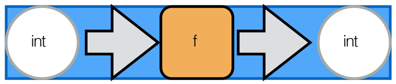
* Before we can use functions, we need to be able to declare them.
* A function declaration has two parts:
* The first part, the *function prototype*, declares the *type signature* (and type) of the function, along with its name.
* The second part, the *function body*, is a block which contains the statements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

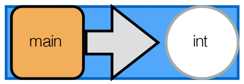
```c
int f(int a);

int main(void) {
  return f(5);
}

int f(int a) {
  int c = a + 1;
  /* more here */
  return c;
}
```

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.

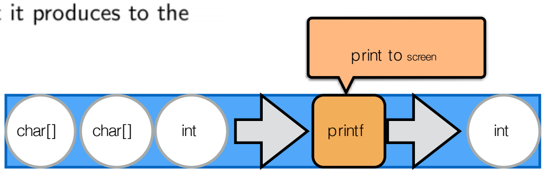
```c
int f(int a);

int main(void) {
  return f(5);
}

int f(int a) {
  int c = a + 1;
  /* more here */
  return c;
}
```





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.



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*.

```
int f(int a);

int main(void) {
  return f(5);
}

int f(int a) {
  int c = a + 1;
  /* more here */
  return c;
}
```

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) {
  return f(5);
}

int f(int a) {
  int c = a + 1;
  /* more here */
  return c;
}
```

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
}
```

Pointers

- As we've just seen, functions *copy* the values in their parameters.
  - They cannot access the variable those values were provided by, directly.
- If we want a function to directly modify the contents of a variable, we need some way to tell the function about the variable itself.
- In C, we do this by telling the function "where in memory we should store values".

Pointer types

- In C, a pointer is a variable that stores the name of location in memory (an "address").
  - Locations in memory are just assigned a number from 0 to some large value.
- Pointers have types, which tell the compiler how we'd like to interpret the value at that address - is it an `int`, a `double`, or whatever.

Declaring pointers

* Much as with arrays, we declare a pointer to a type by "decorating" a name with a symbol.
* For arrays, we had to add [ ] to a name to make it an array of values of that type.
* For pointers, we add a `*` to the start of the name.
* Here `x` and `y` are variables of type `int`. `p` is a variable of type *pointer to int*.

```
int x, *p, y;
```

NULL

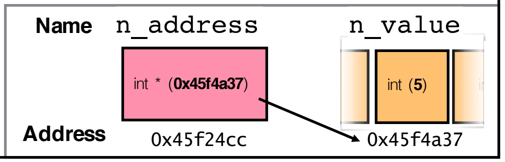
* We can give a pointer variable an initial value, but unlike value types, it is very hard to provide a value that will be useful.
* Clearly, we shouldn't point a pointer at an area of memory that *doesn't* contain a value of the right type!
* The most useful literal pointer value is the special value `NULL`.
* A pointer with the value `NULL` is essentially set to "not point at anything"

```
int x, *p, y;
/*This pointer is
pointing at nothing
explicitly */
int *p1 = NULL;
```

Pointing at a variable

* We will almost always want our pointers to point at the memory used by an existing variable.
* The special operator & provides the address in memory where a variable is located.

```
int n_value = 5;            //declare integer n_value = 5
int *n_address = &n_value;  //declare a pointer n_address
                            // … and give it address of n_value
```



Getting a value from a pointer

* We will almost always want our pointers to point at the memory used by an existing variable.
* The special operator & provides the address in memory where a variable is located.
* The special operator * does the opposite, providing the value located at the address in the pointer.

```
int n_value;
int *n_address = NULL;

n_value = 5;
n_address = &n_value;
n_value = *n_address;
```

Passing pointers to functions

* So, now we can use the & and * operators to allow a function to modify a variable (rather than just use the value in it)
* Here, the function f takes a *pointer to an integer* as its argument.
* It then modifies the value in the location the pointer points at, by adding 2 to it.
* In main, we call f with the *address of* i, so f modifies i's *contents*.
* The value printed is 9, (not 7).

```
void f(int * p)
{
  *p += 2;
}
int main(void){
  int i = 7;
  f(&i);
  printf("%d",i);
}
```

Pointers example

```
#include <stdio.h>

/* Here is a new function prototype */

void f(int * a_ptr); //void f(int *) would also be sufficient

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

    puts("\n Enter an integer");
    fgets(buffer,sizeof(buffer),stdin);
    sscanf(buffer,"%d",&b);

    f(&b); //call f with b's address

    printf("Value in b is now: %d \n\n",b);

    return 0;
    }

void f(int * a_ptr) {
    //explicitly modify value in memory at a_ptr
    *a_ptr += 1;
    //equiv to (*a_ptr)++; or *a_ptr = (*a_ptr) +1;
    //need brackets to ensure we don't actually increment a_ptr itself.
}
```