LECTURE 5:

STRUCTURES

COMP1002J: Introduction to Programming 2

Dr. Brett Becker (brett.becker@ucd.ie)

Beijing Dublin International College

Structures

- Basic data types hold single values (ints, floats, chars, etc.)
- Arrays allow you to combine these basic types into lists – groups of values **of the same type** accessible by an index.
- In the real world things are not always so clear:
  - A person has a name (string) and an age (int)
- A key feature of programming languages is that they allow us to represent and manipulate real world concepts (e.g. bank accounts, music and book collections, student records, people, etc.)
- To enable this, C provides **structures**.

What is a Structure?

- A structure is a collection of variables, known as **members** (data members, fields) that can be of **different data types**.

- Structures are declared outside of functions (they have global scope) using the `struct` keyword.

- **Example**: Date and Time

```
struct Date {
     int day;
     int month;
     int year;
};
```

```
struct Time {
        int hour;
        int minute;
        int second;
};
```

Combining Structures

- Members of a structure can themselves be structures.

```
struct DateTime {
      struct Date date;
      struct Time time;
};
```

- Structures can also contain arrays:

```
struct StudentRecord {
      char name[50];
      int id;
      struct Date dateOfBirth;
};
```

More on this later

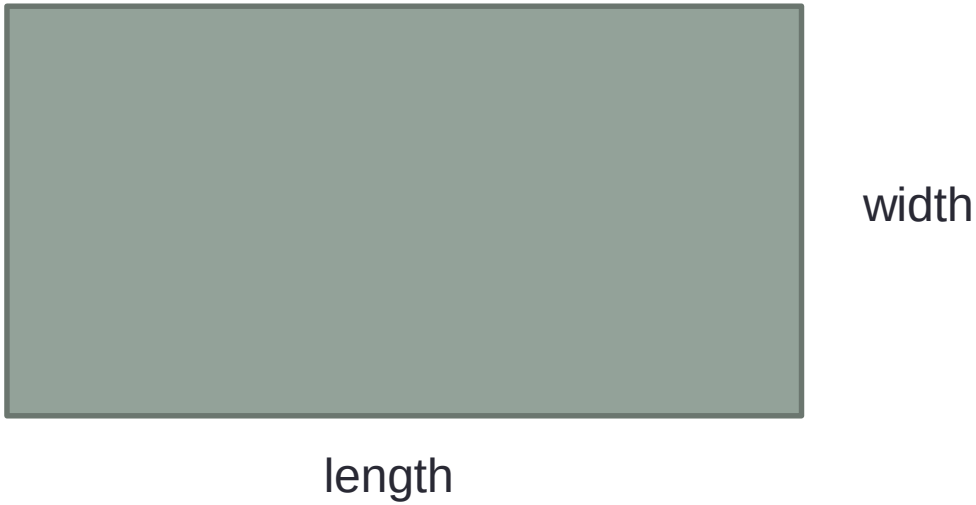
Program Design & Data Modelling

- A key skill in programming is deciding how best to store data in the program.
- Big challenge: understand *what* data needs to be stored and *how* it will be accessed.
- Before we can meet these challenges we need to understand how concepts in the real world can be represented in C.
- The main trick to this is to identify the **defining features** of the concept:
  - For example, the defining features of a **date** are the day, month and year. So three items make a date.
  - The defining feature of a **square** is the length of its sides. By definition, all four sides are the same, so there is only one item.

Problems

- Develop a structure to represent a Rectangle.

- A **rectangle** has a length and a width. So there are two items.



Problems

- Develop a structure to represent a Rectangle.

- A **rectangle** has a length and a width

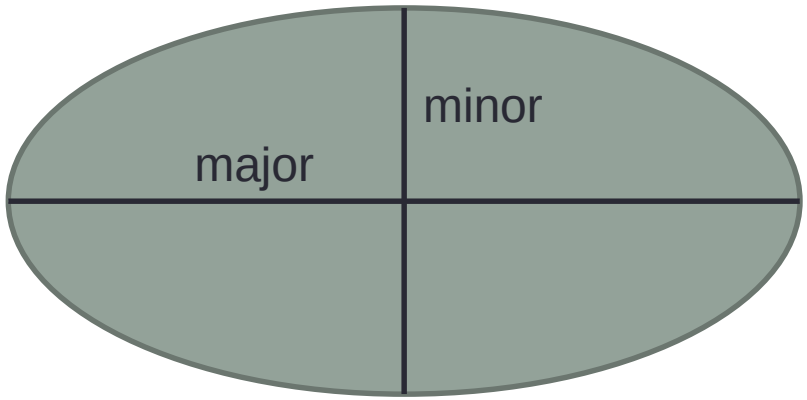
```
struct Rectangle {
    int length;
    int width;
};
```

- Is `int` a good choice? Maybe the rectangle will not have whole-integer lengths. This is for you, the programmer to find out and decide!

Problems

- Develop a structure to represent an Ellipse (Oval).

- An **ellipse** has a major axis and a minor axis. Again, two items.



Problems

- Develop a structure to represent an Ellipse (Oval).

- An **ellipse** has a major axis and a minor axis

```
struct Ellipse {
     float major;
     float minor;
};
```

In General

- In general, we have three broad types of data:
  - *numeric*, where we store numbers using an appropriate data type (int/long/float/double)
  - *textual*, where we store text strings
  - *complex*: made up of other structures

- The general rule of thumb is that we use numeric data types for data we might need to do **calculations** on.
  - Some data that looks numeric is really textual:
    - We don't do mathematical calculations on phone numbers, student IDs, etc.
    - **Therefore the `StudentRecord` structure we saw earlier should be changed so that `id` is not an `int`.**

Problems

- Develop a structure to represent a Book.

- A book has a name, a **list** of authors, a year of publication, a publisher, a number of pages, an ISBN number, a genre, etc.

```
struct Book {
        char name[200];
        char authors[5][100];
        int year;
        char publisher[100];
        int pages;
        char isbn[13];
        char genre[50];
};
```

This is an array of strings. (A two-dimensional array). We will study these more later.

This array can hold 5 strings.

Each string can be up to 99 characters long (remember the last character must be \0)

Problems

- Develop one or more structures to represent a Restaurant Menu

- A restaurant menu consists of a set of *dishes* that are sold by the restaurant.
  - Dishes can be classified as starter, main course, or dessert. Dishes have a name and a price.
  - Dishes may be mild, medium, or hot in terms of spiciness.

Problems

- Develop one or more structures to represent a Restaurant Menu

- A restaurant menu consists of a set of *dishes* that are sold by the restaurant. Dishes can be classified as starter, main course, or dessert. **Dishes have a name and a price. Dishes may be mild, medium, or hot in terms of spiciness.**

```
struct Dish {
      char name[50];
      int price;
      int spiciness; // should be 0,1,or 2
};
```

Problems

- Develop one or more structures to represent a Restaurant Menu

- A restaurant menu consists of a set of *dishes* that are sold by the restaurant. **Dishes can be classified as starter, main course, or dessert.** Dishes have a name and a price. Dishes may be mild, medium, or hot in terms of spiciness.

```
struct Menu {
      struct Dish starters[50];
      struct Dish mains[50];
      struct Dish desserts[50];
};
```

Using Structures

Let's use this structure.

```
struct Time {
    int hour;
    int minute;
    int second;
};
```

Using Structures

- To create a variable that can hold a structure you write the keyword `struct` followed by the **name of the structure** and then the **variable name**:

```
struct Time myTime;
```

`struct` keyword

Name of the structure type

Variable name (The name of *this* structure)

Using Structures

- To access a member of a structure you must use the dot operator (.):

```
variable.member
```

- For example, to assign a value to the hour member of the mytime variable you should use:

```
myTime.hour = 12;
```

- The same notation can be used to read the values stored in the members:

```
printf("%d", myTime.hour);
```

Using Structures

- Just like other variables, this line results in a block of memory being set aside.
  - The amount of memory used by a struct can be checked using the `sizeof` operator.

    `sizeof(struct Time)` is 12 bytes

  - Why? An integer is 4 bytes, and there are 3 int members in the Time struct!

file: time1.c

Using Structures

- What is the size of the book structure?

```
struct Book {
        char name[200];
        char authors[5][100];
        int year;
        char publisher[100];
        int pages;
        char isbn[13];
        char genre[50];
};
```

Sometimes, the result of sizeof() is a few bytes bigger than we expect: C can sometimes add extra bytes for efficiency reasons.

Sometimes, it doesn't.

- What about the Menu structure?

```
struct Menu {
        struct Dish starters[50];
        struct Dish mains[50];
        struct Dish deserts[50];
};
```

```
struct Dish {
        char name[50];
        float price;
        int spiciness;
};
```

file: book.c, menu_dish.c

Initialising Structures

- In cases where you want to declare and initialise a structure variable, you can use special notation to set the values (similar to initialising an array):

```
struct Time lunch = {11, 30, 0};
```

- You can then print out the time as follows:

```
printf("%d:%d:%d", lunch.hour, lunch.minute, lunch.second);
```

- This would print out:

11:30:0

Structures & Pointers

- Pointers can be used with structures:

```
struct Time lunch = {11, 30, 0};
struct Time *time; // a pointer to a Time structure
time = &lunch; // points to the 'lunch' structure
```

- There are two ways to access the members of a structure from a pointer:

```
printf("the hour is: %d", (*time).hour);
printf("the minute is: %d", time->minute);
```

- This second approach uses the **arrow operator** (->)

Structures & Pointers

```
printf("the hour is: %d", (*time).hour);
```

We use brackets here because we want the hour member of the structure that time points to. Without the () we would be asking for a pointer to the member `hour`.

```
printf("the hour is: %d", *time.hour);
```

Structures & Pointers

- General rule:

- If you have a structure, use a dot (.)

```
struct Time lunch = {11, 30, 0};
printf( "Lunch hour: %d\n", lunch.hour );
```

- If you have a pointer to a structure, use an arrow (->)

```
struct Time lunch = {11, 30, 0};
struct Time *lunchptr;
lunchptr = &lunch;
printf( "Lunch hour: %d\n", lunchptr->hour );
```

Example: Time Explorer

```
#include <stdio.h>
struct Time {
        int hour;
        int minute;
        int second;
};
int main() {
        struct Time now, next;
        printf("Please enter the time now (h m s):");
        scanf("%d%d%d", &now.hour, &now.minute, &now.second);

        next.hour = (now.hour + 1) % 24;
        next.minute = now.minute;
        next.second = now.second;

        printf("The time in 1 hour will be: %d:%d:%d\n",
                next.hour, next.minute, next.second);
}
```

Now can you see why we use & with `scanf`?

Answer: we tell it the address to store the value at.

file: timeexplorer.c

Structures as Types

- It is possible to create a *user-defined type* from a structure.
- To do this, we use the typedef keyword:

```
typedef struct {
     int hour;
     int minute;
     int second;
} Time;
…
Time myTime;
```

Example: Time Explorer 2

```
#include <stdio.h>
typedef struct {
        int hour;
        int minute;
        int second;
} Time;
main() {
        Time now, next;
        printf("Please enter the time now (h m s):");
        scanf("%d%d%d", &now.hour, &now.minute, &now.second);

        next.hour = (now.hour + 1) % 24;
        next.minute = now.minute;
        next.second = now.second;

        printf("The time in 1 hour will be: %d:%d:%d",
                next.hour, next.minute, next.second);
}
```

Look at the difference between this and timeexplorer1.c

file: timeexplorer2.c

Structures & Functions

- Structures can be used as parameters in functions:

```
void display_time(Time time) {
    printf("%02d:%02d:%02d", time.hour,
                 time.minute, time.second);
}
```

- You can call the function as follows:

```
Time time = {12,0,0};
display_time(time);
```

- This will print out:

12:00:00

%02d means to print an integer (so we use %d) but to make it at least two digits by putting a 0 before any 1-digit number

file: displaytime.c

Structures & Functions

- BUT…
- Function calls are pass by value – this means that the arguments are copied into a new location in memory!
- This can have a massive impact on performance, which in turn can affect things like speed and battery life.
- Also, this means that you cannot change the values stored in a structure inside the function (just like the basic data types).
- But if some members are pointers, we don’t copy all the data, and the data they point to **can be modified** from within the function!

Structures & Functions

- We generally prefer to use pass-by-reference for passing structures into functions.
  - Pass in the address of the structure.
  - Function parameter is a pointer.
- Why?
  - No need to create a copy of the structure – saves time and memory.
  - The structure can be changed inside the function.

Structures & Functions

- Example: add_hour – adds an hour to a time:

```
void add_hour( Time *time ) {
   time->hour = ( time->hour + 1 ) % 24;
}

// then we can call it like this:
Time t = { 12, 30, 00 };
add_hour( &t );
```

file: roll_time_value.c,
roll_time_reference.c