Identifying related data

Let's consider the 2012 1st project for CITS1002.

The goal of the project was to manage the statistics of AFL teams throughout the season, calculating their positions on the premiership ladder at the end of each week.

Let's consider the significant global variables in its sample solution:

```c
// DEFINE THE LIMITS ON PROGRAM'S DATA-STRUCTURES
#define MAX_TEAMS 24
#define MAX_TEAMNAME_LEN 30

// DEFINE A 2-DIMENSIONAL ARRAY HOLDING OUR UNIQUE TEAMNAMES
char teamname[MAX_TEAMS][MAX_TEAMNAME_LEN+1]; // +1 for null-byte

// STATISTICS FOR EACH TEAM, INDEXED BY EACH TEAM'S 'TEAM NUMBER'
int played[MAX_TEAMS];
int won[MAX_TEAMS];
int lost[MAX_TEAMS];
int draw[MAX_TEAMS];
int bagass[MAX_TEAMS];
int points[MAX_TEAMS];

// PRINT EACH TEAM'S RESULTS, ONE-PER-LINE, IN NO SPECIFIC ORDER
for(int t=0 ; t<ntteams ; ++t)
{
    printf("%s %i %i %i %i %.2f \n", teamname[t],
    played[t], won[t], lost[t], draw[t],
    bagass[t], points[t]); // calculate percentage
}
```

It's clear that the variables are all strongly related, but that we're naming and accessing them as if they are independent.
Defining structures

Instead of storing and identifying related data as independent variables, we prefer to “collect” it all into a single structure.

C provides a mechanism to bring related data together, structures, using the struct keyword.

We can now define and gather together our related data with:

```c
// DEFINE AND INITIALIZE ONE VARIABLE THAT IS A STRUCTURE
struct
{
    char *name; // a pointer to a sequence of characters
    int red;   // in the range 0..255
    int green;
    int blue;
} rgb_colour = {
    "DodgerBlue",
    30,
    144,
    255
};
```

We now have a single variable (named rgb_colour) that is a structure, and at its point of definition we have initialised each of its 4 fields.
Defining and array of structures

Returning to our AFL project example, we can now define and gather together its related data with:

```c
// DEFINE THE LIMITS ON PROGRAM'S DATA-STRUCTURES
#define MAX_TEAMS       24
#define MAX_TEAMNAME_LEN 30

struct
{
  char  teamname[MAX_TEAMNAME_LEN+1];  // +1 for null-byte
  int   played;
  int   won;
  int   lost;
  int   drawn;
  int   before;
  int   against;
  int   points;
} team[MAX_TEAMS];  // DEFINE A 1-DIMENSIONAL ARRAY NAMED team
```

We now have a single (1-dimensional) array, each element of which is a structure. We often term this an array of structures.

Each element of the array has a number of fields, such as its teamname (a whole array of characters) and an integer number of points.
Accessing the fields of a structure

Now, when referring to individual items of data, we need to first specify which team we're interested in, and then which field of that team's structure.

We use a single dot ("." or fullstop) to separate the variable name from the field name.

The old way, with independent variables:

```c
// PRINT EACH TEAM'S RESULTS, ONE-PER-LINE, IN NO SPECIFIC ORDER
for(int t=0 ; t<teams ; ++t)
{
    printf("%s %i %i %i %i %.2f %i\n", // %age to 2 decimal-places
            teamname[t],
           played[t], won[t], lost[t], drawn[t],
           baglost[t], bagwin[t],
           (100.0 * baglost[t] / bagwin[t]), // calculate percentage
           points[t]);
}
```

The new way, accessing fields within each structure:

```c
// PRINT EACH TEAM'S RESULTS, ONE-PER-LINE, IN NO SPECIFIC ORDER
for(int t=0 ; t<teams ; ++t)
{
    printf("%s %i %i %i %i %.2f %i\n", // %age to 2 decimal-places
            team[t].teamname,
           team[t].played, team[t].won, team[t].lost, team[t].drawn,
           team[t].baglost, team[t].bagwin,
           (100.0 * team[t].baglost / team[t].bagwin), // calculate percentage
           team[t].points);
}
```

While it requires more typing(!), it's clear that the fields all belong to the same structure (and thus team). Moreover, the names `teamname, played, ....` may now be used as "other" variables elsewhere.
**Accessing system information using structures**

Operating systems (naturally) maintain a lot of (related) information, and keep that information in structures.

So that the information about the structures (the datatypes and names of the structure's fields) can be known by both the operating system and users' programs, these structures are defined in system-wide header files—typically in /usr/include and /usr/include/sys.

For example, consider how an operating system may represent time on a computer:

```c
#include <stdio.h>
// A value accurate to the nearest microsecond but also has a range of years
struct timeval
{
    int tv_sec;      // Seconds
    int tv_usec;     // Microseconds
};
```

Note that the structure has now been given a name, and we can now define multiple variables having this named datatype (in our previous example, the structure would be described as *anonymous*).

We can now request information from the operating system, with the information returned to us in structures:

```c
#include <stdio.h>
#include <sys/time.h>
    struct timeval start_time;
    struct timeval stop_time;
    gettimeofday( &start_time, NULL );
    printf("program started at %1.6f\n", (int)start_time.tv_sec, (int)start_time.tv_usec);
    ....
    perform_work();
    ....
    gettimeofday( &stop_time, NULL );
    printf("program stopped at %1.6f\n", (int)stop_time.tv_sec, (int)stop_time.tv_usec);
```

Here we are passing the structure by address with the & operator, so that the `gettimeofday()` function can modify the fields of our structure. (we're not passing a meaningful pointer as the second parameter to `gettimeofday()`, as we're not interested in timezone information)
Accessing structures using a pointer

We've seen that we can access fields of a structure using a single dot (\'.\' or fullstop).
What if, instead of accessing the structure directly, we only have a pointer to a structure?
We've seen "one side" of this situation, already - when we passed the address of a structure to a function:

```c
struct timeval start_time;
gettimeofday( &start_time, NULL );
```

The function `gettimeofday()`, must have been declared to receive a pointer:

```c
extern int gettimeofday( struct timeval *time, ......);
```

Consider the following example, in which a pointer to a structure is returned from a function.
We now use the \rightarrow operator (pronounced the 'arrow', or 'points-to' operator) to access the fields via the pointer:

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

void greeting(void)
{
    time_t NOW = time(NULL);
    struct tm *tm = localtime(&NOW);
    printf("Today's date is %l/%l/%l\n",
            tm->tm_mday, tm->tm_mon + 1, tm->tm_year + 1900);
    if(tm->tm_hour < 12)
    {
        printf("Good morning\n");
    }
    else if(tm->tm_hour < 17)
    {
        printf("Good afternoon\n");
    }
    else
    {
        printf("Good evening\n");
    }
}
```
Defining our own datatypes

We can further simplify our code, and more clearly identify related data by defining our own datatypes.

We use the `typedef` keyword to define our new datatype in terms of an old (existing) datatype.

```c
#define MAX_TEAMS 24
#define MAX_TEAMNAME_LEN 30

typedef struct {
    char tablename[MAX_TEAMNAME_LEN+1]; // +1 for null-byte
    int played;
    ...
} TEAM;
TEAM team[MAX_TEAMS];
```

As a convention (but not a C99 requirement), we'll define our user-defined types using uppercase names.

```c
#define PRINT EACH TEAM'S RESULTS, ONE-PER-LINE, IN NO SPECIFIC ORDER
for(int t=0; t<nteams; ++t) {
    TEAM *tp = &team[t];
    printf("%s %i %i %i %i %.2f %i\n", // %age to 2 decimal-places
        tp->tablename,
        tp->played, tp->won, tp->lost, tp->drawn,
        tp->bfog, tp->bagainst,
        (100.0 * tp->bfog / tp->bagainst), // calculate percentage
        tp->points);
}
```

---

Another example - using a pointer to our own datatype

Let's consider another example - the starting (home) and ending (destination) bus stops from the CITS2002 1st project of 2015.

We start with some of its definitions:

```c
// GLOBAL CONSTANTS; BEST DEFINED ONCE NEAR THE TOP OF FILE
#define MAX_STOPS_NEAR_ANYWHERE 200  // in Trapanth: 184

// 2-D ARRAY OF VIABLE STOPs FOR COMMENCEMENT OF JOURNEY
char viable_home_stpid[MAX_STOPS_NEAR_ANYWHERE][MAX_FIELD_LEN];
char viable_home_name[MAX_STOPS_NEAR_ANYWHERE][MAX_FIELD_LEN];
int viable_home_metres[MAX_STOPS_NEAR_ANYWHERE];
int n_viable_homes = 0;

// 2-D ARRAY OF VIABLE STOPs FOR END OF JOURNEY
char viable_dest_stpid[MAX_STOPS_NEAR_ANYWHERE][MAX_FIELD_LEN];
char viable_dest_name[MAX_STOPS_NEAR_ANYWHERE][MAX_FIELD_LEN];
int viable_dest_metres[MAX_STOPS_NEAR_ANYWHERE];
int n_viable_dests = 0;
```

(After a post-project workshop) we later modified the 2-dimensional arrays to use dynamically-allocated memory:

```c
// 2-D ARRAY OF VIABLE STOPs FOR COMMENCEMENT OF JOURNEY
char **viable_home_name = NULL;
int *viable_home_metres = NULL;
int n_viable_homes = 0;

// 2-D ARRAY OF VIABLE STOPs FOR END OF JOURNEY
char **viable_dest_name = NULL;
int *viable_dest_metres = NULL;
int n_viable_dests = 0;
```

and we can now use `typedef` to define our own datatype:

```c
// A NEW DATATYPE TO STORE 1 VIABLE STOP
typedef struct {
    char *stpid;
    char *name;
    int metres;
} VIABLE;

// A VECTOR FOR EACH OF THE VIABLE home AND dest STOPs
VIABLE *home_stops = NULL;
VIABLE *dest_stops = NULL;
int n_home_stops = 0;
int n_dest_stops = 0;
```
Finding the attributes of a file

The operating system manages its data in a file system, in particular maintaining its files in a hierarchical directory structure - directories contain files and other (sub)directories.

As we saw with time-based information, we can ask the operating system for information about files and directories, by calling some system-provided functions.

We employ another POSIX\(^\dagger\) function, `stat()`, and the system-provided structure `struct stat`, to determine the attributes of each file:

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <time.h>
#include <unistd.h>

char *progame;

void file_attributes(char *filename)
{
    struct stat stat_buffer;
    if(stat(filename, &stat_buffer) != 0) // can we 'stat' the file's attributes?
    {
        perror(progame);
        exit(EXIT_FAILURE);
    }
    else if( S_ISREG(stat_buffer.st_mode ) )
    {
        printf( "%s is a regular file\n", filename );
        printf( "is %li bytes long\n", stat_buffer.st_size );
        printf( "and was last modified on %li\n", stat_buffer.st_mtime );
        printf( "which was %s", ctime( &stat_buffer.st_mtime ) );
    }
}
```

\(^\dagger\)POSIX is an acronym for "Portable Operating System Interface", a family of standards specified by the IEEE for maintaining compatibility between operating systems. POSIX defines the application programming interface (API), along with command line shells and utility interfaces, for software compatibility with variants of Unix (such as OS-X and Linux) and other operating systems (e.g. Windows has a POSIX emulation layer).
Reading the contents of a directory

Most modern operating systems store their data in hierarchical file systems, consisting of directories which hold items that, themselves, may either be files or directories.

The formats used to store information in directories in different file-systems are different(!), and so when writing portable C programs, we prefer to use functions that work portably.

Consider the strong similarities between opening and reading a (text) file, and opening and reading a directory:

```c
#include <stdio.h>

void print_file(char *filename)
{
    FILE *fp;
    char line[BUFSIZ];
    fp = fopen(filename, "r");
    if(fp == NULL)
    {
        perror("proname");
        exit(EXIT_FAILURE);
    }
    while(fgets(line, sizeof(buf), fp) != NULL)
    {
        printf("%s", line);
    }
    fclose(fp);
}
```

```c
#include <stdio.h>
#include <sys/types.h>
#include <dirent.h>

void list_directory(char *dirname)
{
    DIR *dirp;
    struct dirent *dp;
    dirp = opendir(dirname);
    if(dirp == NULL)
    {
        perror("proname");
        exit(EXIT_FAILURE);
    }
    while((dp = readdir(dirp)) != NULL)
    {
        printf("%s\n", dp->d_name);
    }
    closedir(dirp);
}
```

With directories, we're again discussing functions that are not part of the C99 standard, but are defined by POSIX standards.
Investigating the contents of a directory

We now know how to open a directory for reading, and to determine the names of all items in that directory.

What is each "thing" found in the directory - is it a directory, is it a file...

To answer those questions, we need to employ the POSIX function, stat(), to determine the attributes of the items we find in directories:

```c
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <dirent.h>
#include <unistd.h>

static void list_directory(char *dirname)
{
    char fullpath[MAXPATHLEN];

    while((dp = readdir(dirp)) != NULL)
    {
        struct stat *stat_buffer;

        sprintf(fullpath, "%s/%s", dirname, dp->d_name);
        if(stat(fullpath, &stat_buffer) != 0)
            perror( progname );
        else if( S_ISDIR( stat_buffer.st_mode ))
            printf( "%s is a directory\n", fullpath );
        else if( S_ISREG( stat_buffer.st_mode ))
            printf( "%s is a regular file\n", fullpath );
        else
            printf( "%s is unknown\n", fullpath );
    }
    closedir(dirp);
}
```

```c
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <dirent.h>
#include <unistd.h>

static void list_directory(char *dirname)
{
    char fullpath[MAXPATHLEN];

    while((dp = readdir(dirp)) != NULL)
    {
        struct stat *stat_buffer;
        struct *pointer = &stat_buffer;

        sprintf(fullpath, "%s/%s", dirname, dp->d_name);
        if(stat(fullpath, pointer) != 0)
            perror( progname );
        else if( S_ISDIR( pointer->st_mode ))
            printf( "%s is a directory\n", fullpath );
        else if( S_ISREG( pointer->st_mode ))
            printf( "%s is a regular file\n", fullpath );
        else
            printf( "%s is unknown\n", fullpath );
    }
    closedir(dirp);
}
```