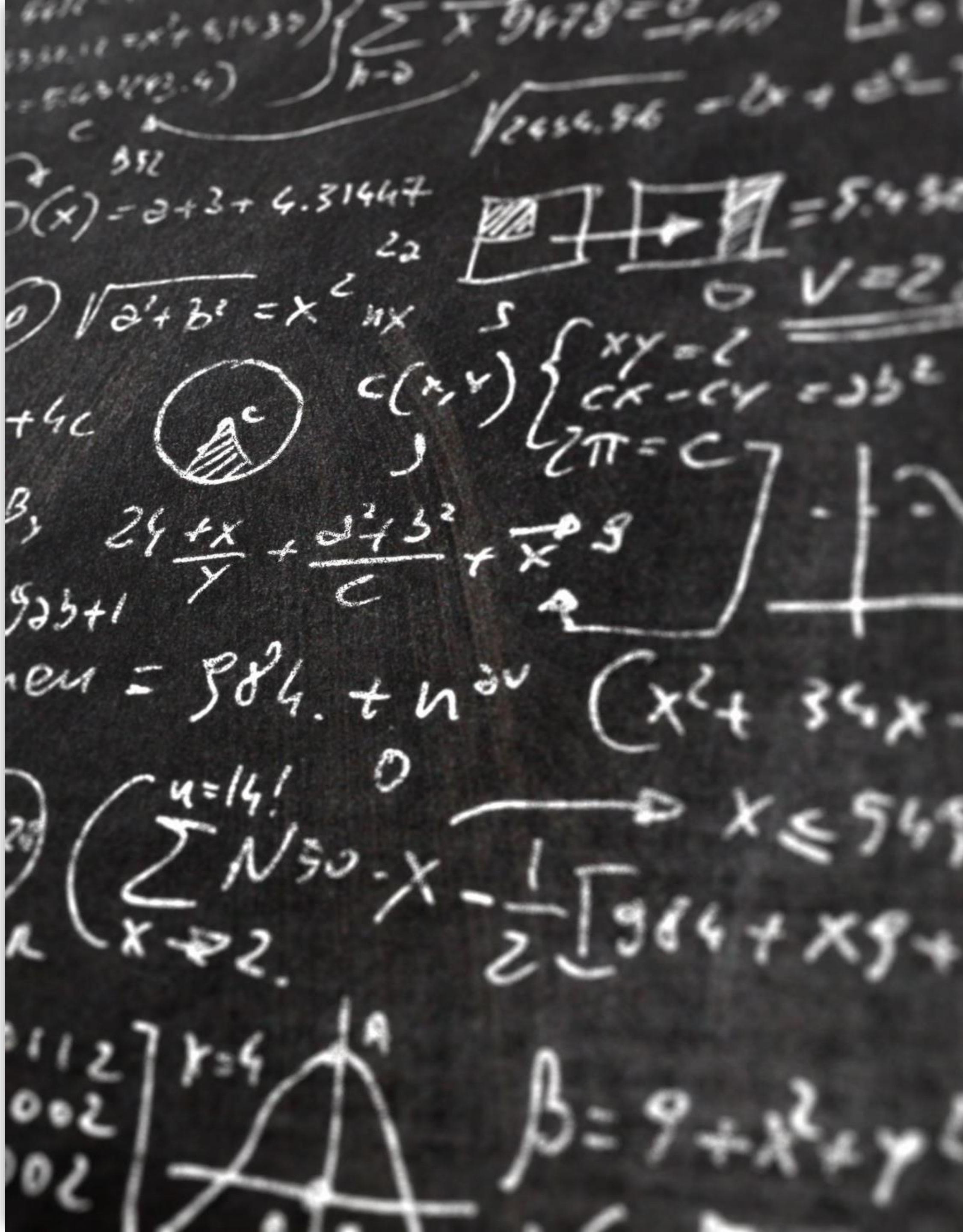
SCC.111 Software Development – Lecture Lecture 19: C highlights

Adrian Friday, Nigel Davies, Hansi Hettiarachchi, Saad Ezzini

This lecture

- Review some of the interesting C features of lab exercise solutions
- Using compound types and pointers to make I/O easier!
- Focus on tradeoffs and style

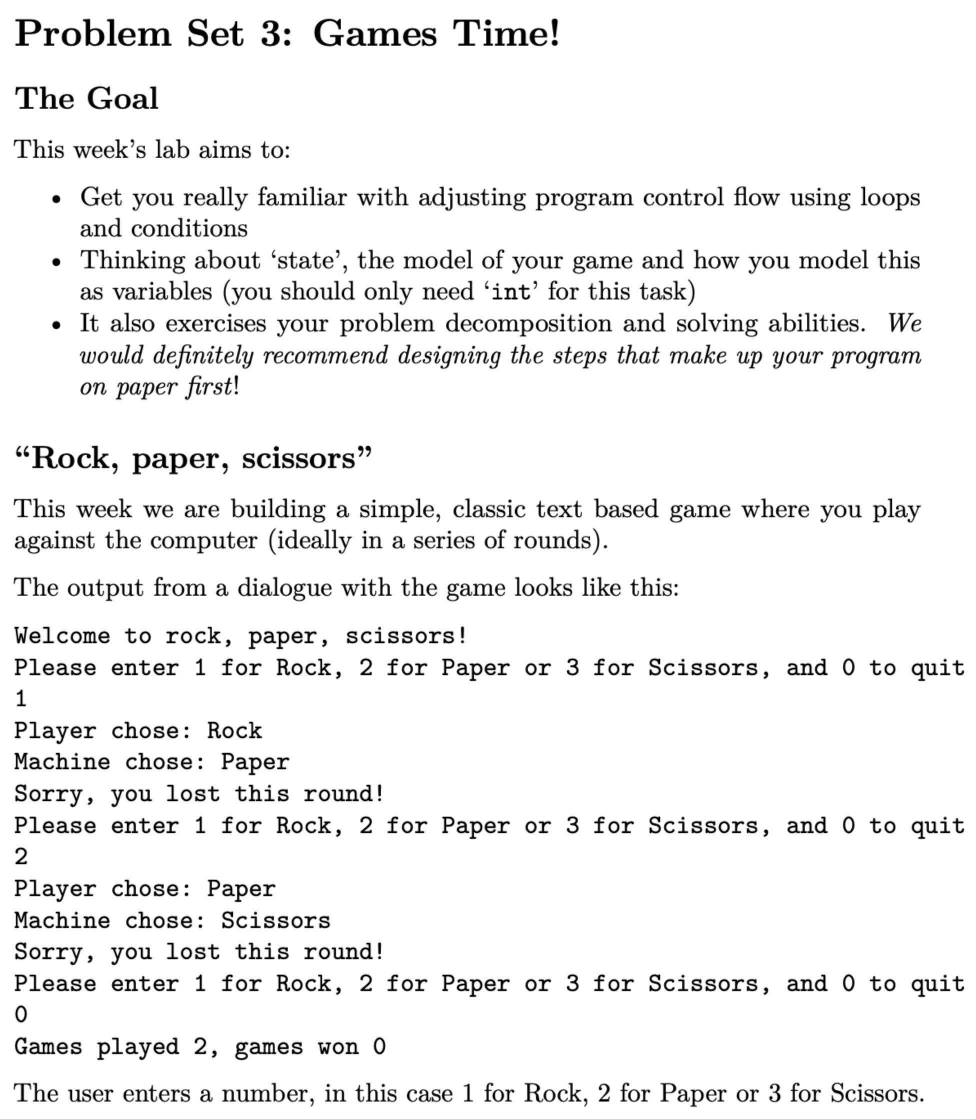


Everybody should learn to program a computer because it teaches you how to think.

Steve Jobs

We set problems to exercise what you’ve learnt…

Rock, paper, scissors… [lizard, spock]



Problem Set 3: Games Time!

The Goal

This week's lab aims to:

- Get you really familiar with adjusting program control flow using loops and conditions
- Thinking about 'state', the model of your game and how you model this as variables (you should only need '`int`' for this task)
- It also exercises your problem decomposition and solving abilities. *We would definitely recommend designing the steps that make up your program on paper first*!

"Rock, paper, scissors"

This week we are building a simple, classic text based game where you play against the computer (ideally in a series of rounds).

The output from a dialogue with the game looks like this:

```
Welcome to rock, paper, scissors!
Please enter 1 for Rock, 2 for Paper or 3 for Scissors, and 0 to quit
1
Player chose: Rock
Machine chose: Paper
Sorry, you lost this round!
Please enter 1 for Rock, 2 for Paper or 3 for Scissors, and 0 to quit
2
Player chose: Paper
Machine chose: Scissors
Sorry, you lost this round!
Please enter 1 for Rock, 2 for Paper or 3 for Scissors, and 0 to quit
0
Games played 2, games won 0
```

The user enters a number, in this case 1 for Rock, 2 for Paper or 3 for Scissors.

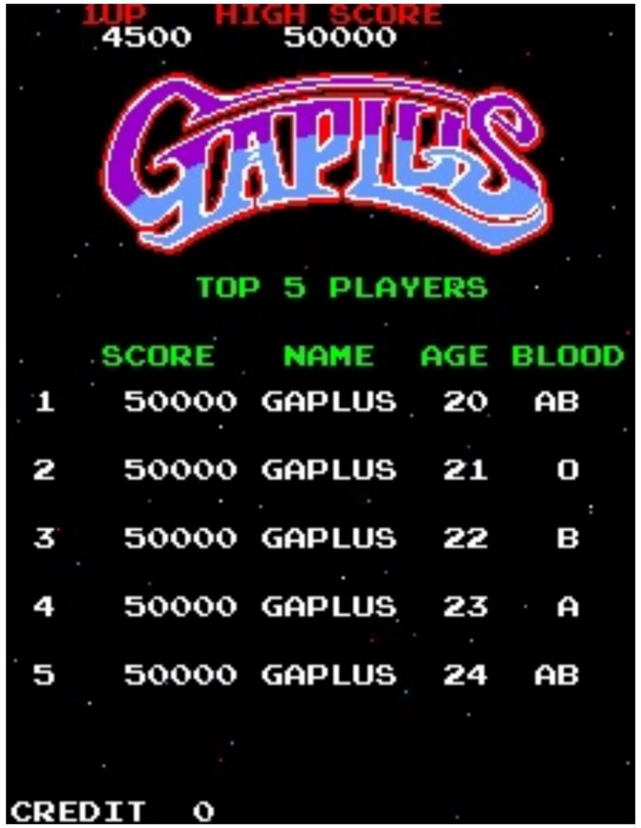
Interesting feature 1 – indexing O(1) lookup

```
char *stringVersion[] = {"Rock", "Paper", "Scissors"};

/* Machine picks their choice */

int machineChoice = rand() % 3 + 1;

printf("Player chose: %s\n",
	stringVersion[playerChoice - 1]);
```

Interesting feature 2 – deciding who won (if or switch?)

```
/* Calculate who won */

bool victory = false;

switch (playerChoice)
{
  case ROCK:
    victory = machineChoice == SCISSORS;
    break;
  case PAPER:
    victory = machineChoice == ROCK;
    break;
    case SCISSORS:
      victory = machineChoice == PAPER;
}
```

High score table


1UP 4500 HIGH SCORE 50000

TOP 5 PLAYERS

	SCORE	NAME	AGE	BLOOD
1	50000	GAPLUS	20	AB
2	50000	GAPLUS	21	O
3	50000	GAPLUS	22	B
4	50000	GAPLUS	23	A
5	50000	GAPLUS	24	AB

CREDIT 0

You are tasked with creating a high score table to hold the top 10 scores of some ficticious game. Ideally, your solution should add a score to the table, maintaining the ordering property that the highest score is at the top, in descending order to the lowest of the 10 highest scores last.

The output from my program looked like this by the end of the task:

```
Enter high score (0 to exit):400
*** HIGH SCORE TABLE ***
 1 - 500
 2 - 400
```

Where to store a score?

- Ideally,
  - Array of scores
  - No ‘gaps’
  - Only keep the top 10 scores
  - Keep in order (*strategies?*)

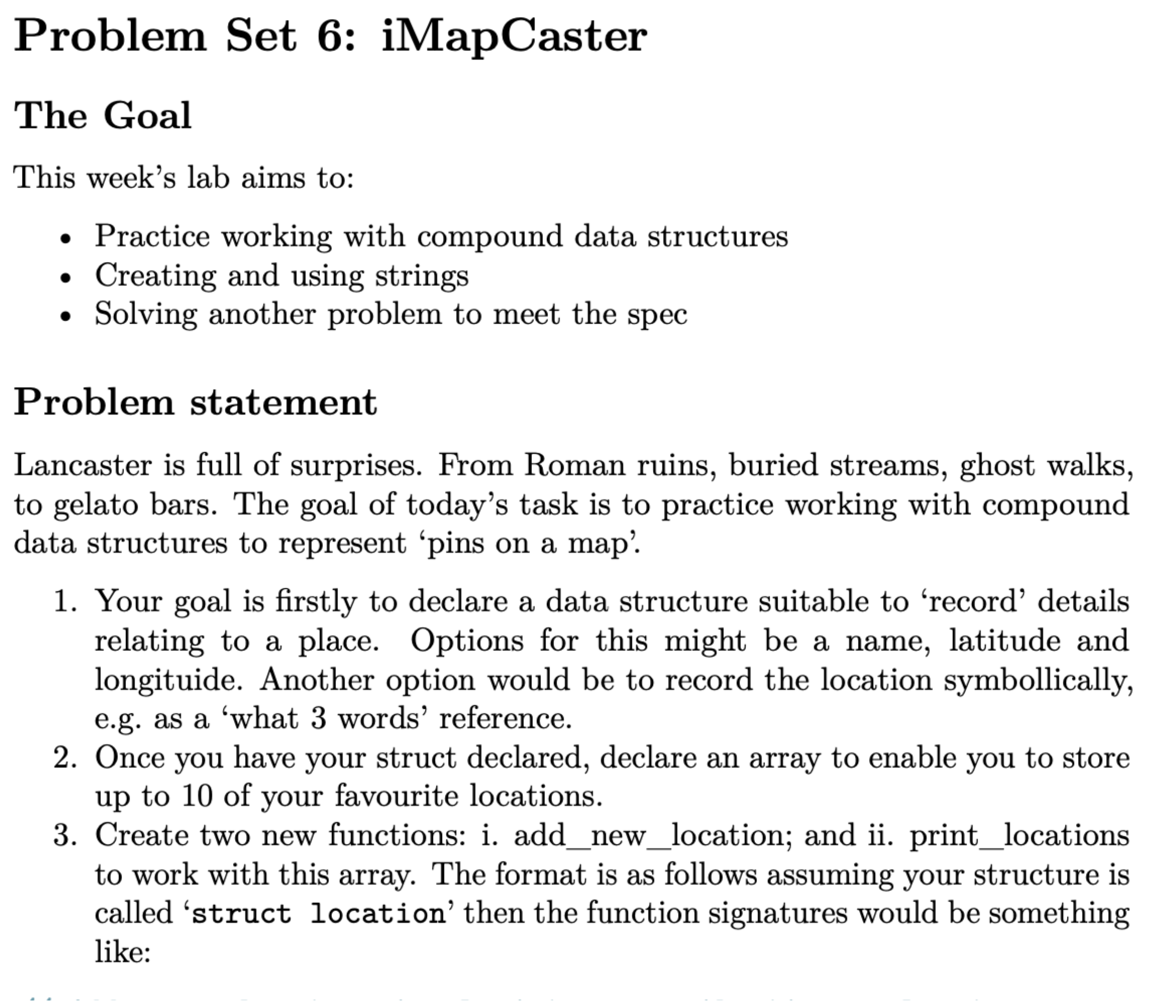
Interesting feature 1 – finding a gap

```
int highscores[MAX_SCORES] = { 0 };

int newScore = -1,
    scoresEntered = 0;

while (newScore != 0) {
  printf("Enter high score (0 to exit):");
  scanf("%d", &newScore);

  if (newScore > 0)
    if (scoresEntered < MAX_SCORES)
      // Room left in table, store it

      highscores[scoresEntered++] = newScore;
```

Interesting feature 2 – shift (backwards) and insert

```
int insertPos = 0;

// Look until we find a score we're greater than

for (; insertPos < maxInTable && newScore <= highscores[insertPos]; insertPos++)
  ;

if (insertPos == maxInTable)
  // Got to end, score was not high enough to register
  return false;

  // Make room for this score

  for (int shiftIndex = maxInTable - 1; shiftIndex > insertPos; shiftIndex--)
    highscores[shiftIndex] = highscores[shiftIndex - 1];

  // Insert new score

  highscores[insertPos] = newScore;
```

Location, location...



Problem Set 6: iMapCaster

The Goal

This week's lab aims to:

- Practice working with compound data structures
- Creating and using strings
- Solving another problem to meet the spec

Problem statement

Lancaster is full of surprises. From Roman ruins, buried streams, ghost walks, to gelato bars. The goal of today's task is to practice working with compound data structures to represent 'pins on a map'.

1. Your goal is firstly to declare a data structure suitable to 'record' details relating to a place. Options for this might be a name, latitude and longituide. Another option would be to record the location symbollically, e.g. as a 'what 3 words' reference.
2. Once you have your struct declared, declare an array to enable you to store up to 10 of your favourite locations.
3. Create two new functions: i. add_new_location; and ii. print_locations to work with this array. The format is as follows assuming your structure is called '`struct location`' then the function signatures would be something like:

Storing locations… as structs

- Representing a location

Representing locations

```
#define FALSE 0
#define TRUE 1

struct location
{
  float latitude, longitude;
  char placename[20];
};

struct location map[10];
```

Interesting feature 1 – dereference then add

```
int add_new_location(char *name, float lat, float lon,
                     struct location *list, int *numLocations)
{
  if (*numLocations < 10) {
    strcpy(list[*numLocations].placename, name);
    list[*numLocations].latitude = lat;
    list[*numLocations].longitude = lon;

    (*numLocations)++;

    return TRUE;
  }

  return FALSE;
}
```

Interesting feature 2 – printing locations (using a pointer to walk an array)

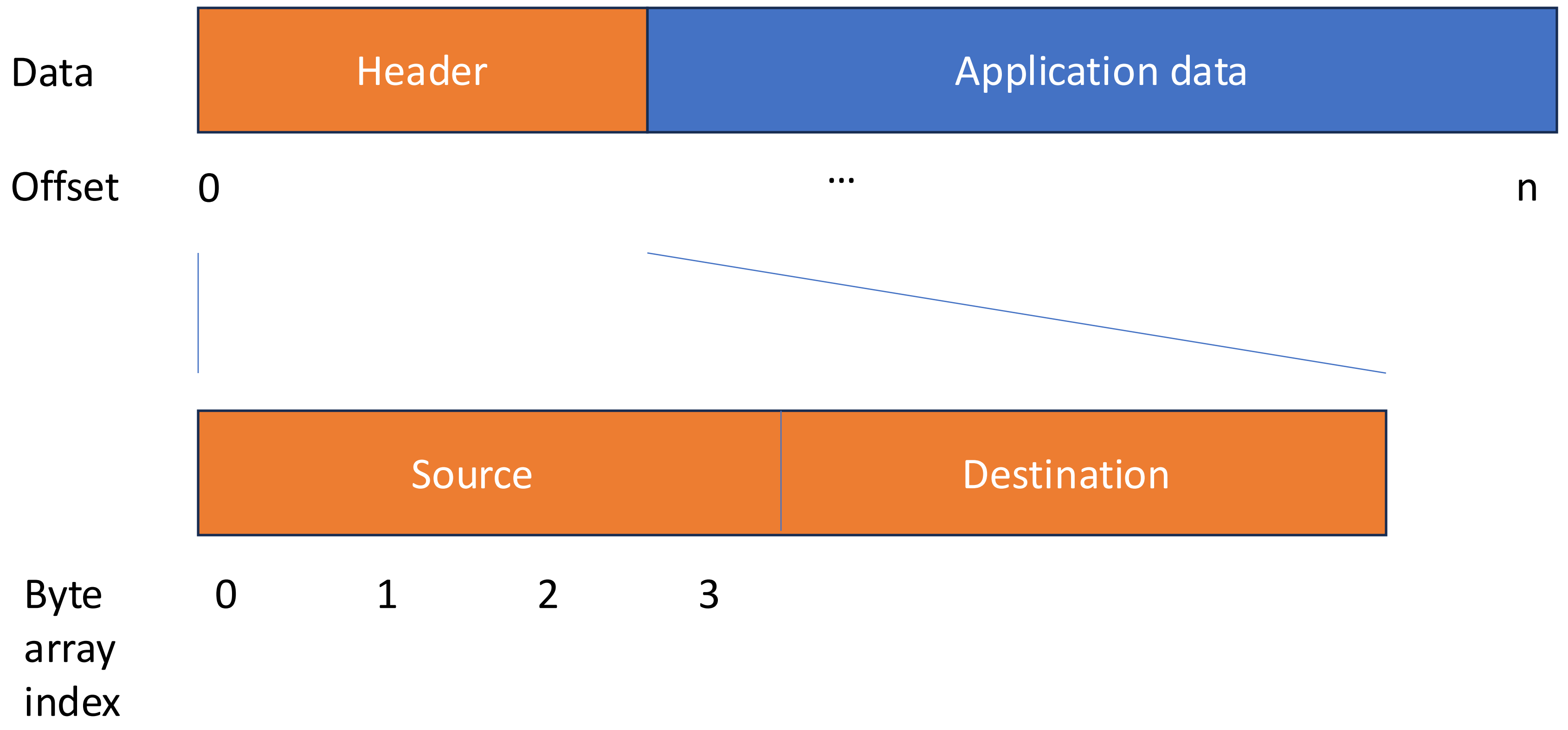
```
void print_locations(struct location *list,
                     int numLocations)
{
  for (int index = 0; index < numLocations; index++)
  {
    struct location *p = &list[index];
    printf("Place %d is %s at %f, %f\n",
           index + 1, p->placename, p->latitude,
           p->longitude);
  }
}
```

Mini case study: buffers and typed pointers

- Consider this:
  - You want to build a **firewall** for your home router OS
  - A firewall's job is to inspect each packet coming in, and decide to **forward** or **drop** (based on **rules**)
  - You need to inspect each packet (data) and look at it's **destination address**

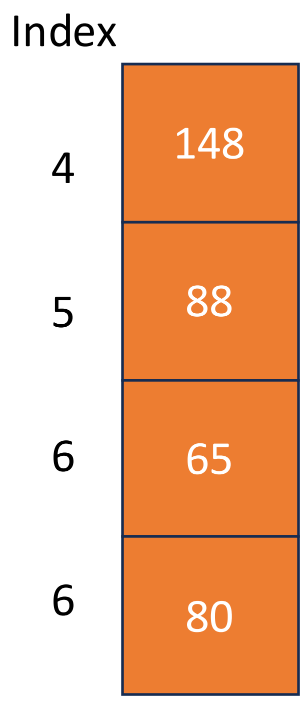
A network packet

- A 'datagram' or buffer of bytes 'with meaning' (binary format)



Accessing the network buffer

- Getting the destination address (4 bytes)

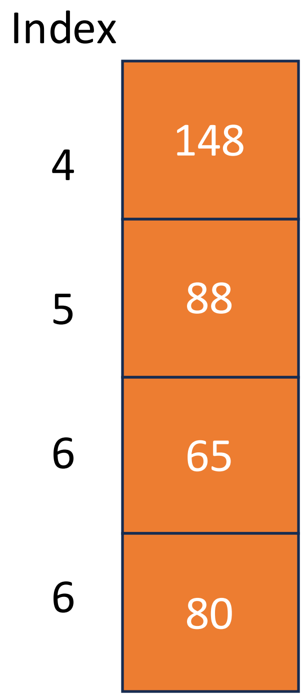


```
int dest = netPacket[4] << 24 |
          netPacket[5] << 16 |
          netPacket[6] << 8 |
          netPacket[7];
```

Get index [4], shift left 24 bits (3 bytes), bitwise OR with….

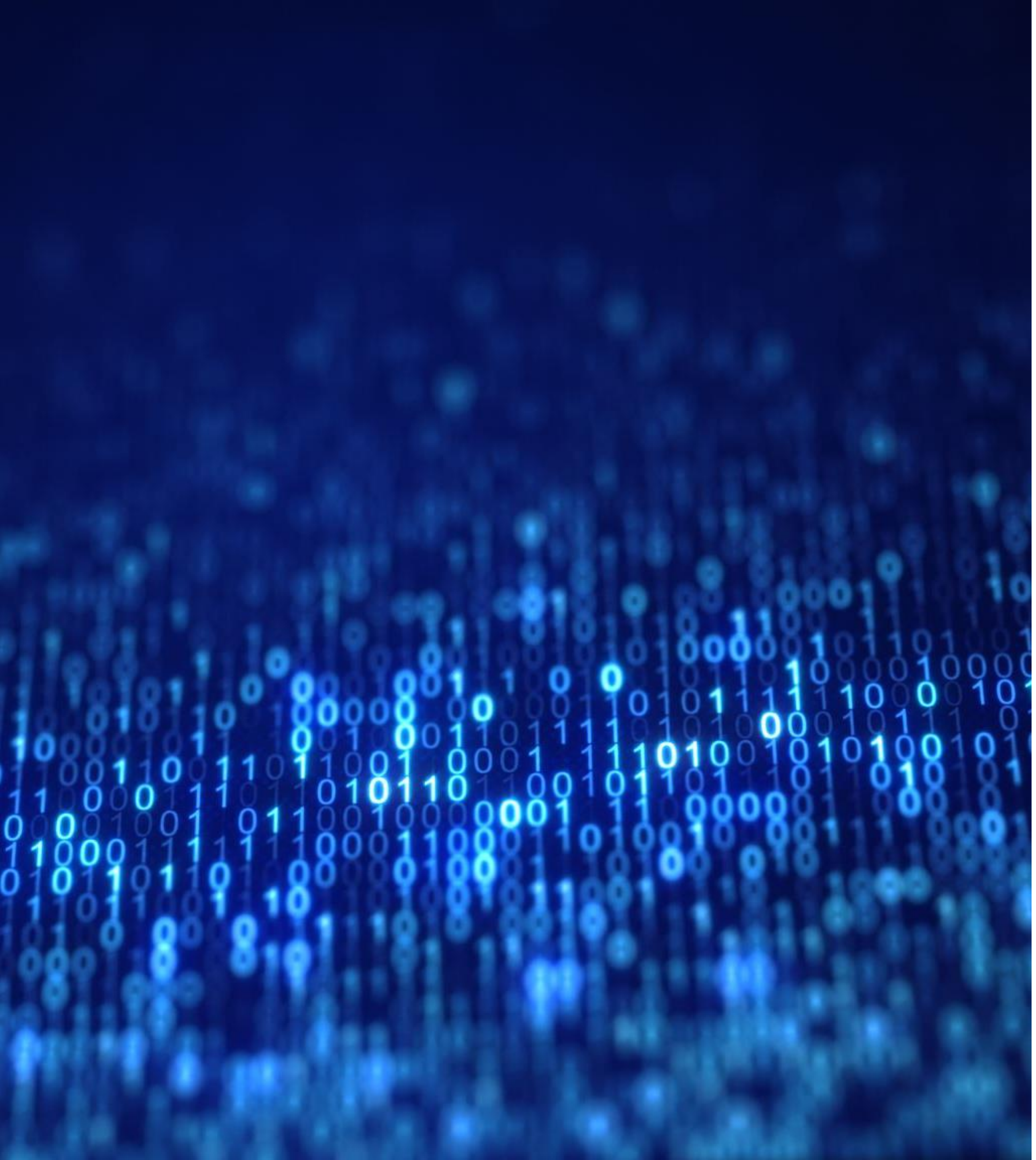
Or

- Getting the destination address (4 bytes)



```
typedef struct tagHeader
{
  int src, dst, len;
  char *payload;
} tPktHdr;

tPktHdr *pkt = (tPktHdr *) netPacket;

printf("Destination address in hex %x\n",
      ntohl(pkt->dst));
```

Cast our header to a new struct (*using pointer magic!*). Note need to respect how integers are stored in our processor (ntohl)



Summary

- Looked at some solutions to weekly exercises focusing particularly on ’special C language features’
- Plus concepts and approaches to data handling using structs, pointers and casts