1.40 Embedded programming in C

(Focus is on small embedded systems such as those based on PIC microcontroller)

EE302 – Real time and embedded systems

1

Overview

- Aims
  - To introduce the issues, tricks, and general approach to writing embedded C applications (focused towards the PIC microcontroller and similar systems)
  - Revise C features used in labs and projects; learn some new C coding tips and tricks
- Learning outcomes – you should be able to…
  - Write a structured pseudocode embedded application outline
  - Give examples and consequences of the constraints which affect embedded applications
  - Give examples of unique features required for embedded C
  - Write a well structured outline C file, suitable for an embedded application
  - Use bit masks and perform bit level manipulations

2

Embedded Application Architecture

3

- In this course we use the Superloop architecture
  - one flow of control in the app.
  - Just one loop in main and this is the superloop
  - All tasks/jobs to be performed are called from this superloop
  - All timing is based on the superloop period
- Other architectures possible (e.g. multi-tasking using a real time operating system)

```
main():
  setup()

  for (all time)
    loop()
```

If we don't show the main function you can assume this form

```
setup():
  …

loop():
  do first thing
  do second thing
  etc…
  delay if required
```

Superloop example pseudocode using Arduino style setup and loop functions. We don't have to use these function names but it is convenient to write this aspect of the code similarly to Arduino.

3

Superloop design guidelines

4

- Once we have decided to use the superloop architecture, designing the programme mostly involves designing the superloop body
- The basic strategy is to divide the superloop body (i.e. the loop function) into the following 3 steps
  - **Input (including sensing and events)**
  - **Processing**
  - **Output (including actuation and side-effects)**
  - We refer to these steps as **Input-process-output** or **sense-think-act**
- System setup (i.e. the setup function) should be implemented according to the needs of the superloop body

4

Contd.

- Input/sensing/events
  - In general input involves anything that originates outside the system and needs to be read or detected
  - E.g. button press, analogue values read via Analogue to Digital Converter (ADC) peripheral, the receiving side of (serial) communications, etc.
- Processing
  - Any transformation of the input (e.g. converting between the raw numeric representation – obtained using the ADC – of temperature sensor's analogue voltage level into a temperature in celcius)
  - Any calculations using the input data and/or "state" of the programme
- Output/actuation/side-effects
  - Providing output to something external to the system, e.g. lighting a LED (digital on/off), actuating a fading LED or a motor drive (PWM output), transmitting (serial) communications, etc.
  - Side effects such as delaying for a certain period of time

Outline your app in pseudocode first

- Try to write the main steps/operations of the program in structured natural language
- Conventions
  - Use indenting (no braces) to indicate operations that should be grouped together or performed within a loop
  - Use double indent for continuation lines if necessary
  - Use self explanatory names
    - Verbs for callable things like functions
    - Nouns for variables

```
loop():
    temp = getProcessTemp()
    if temp outside limits
      shutdownProcess()
    if commsLink ready AND
        data in queue
      sendNextData()
    delay 20 ms


getProcessTemp()   …
shutdownProcess()  …
sendNextData()   …
```

Input/sense
Process/think
Output/act

Process/think
Output/act
Output/act

Pseudocode example

To write the app pseudocode

- Start with the superloop outline and identify the tasks that need to be performed on every loop
  - Keep the superloop itself simple and small
  - **most code should be in functions called by superloop, not superloop itself**

```
loop():
    temp = getProcessTemp()
    if temp outside limits
      shutdownProcess()
    if commsLink ready AND data in queue
      sendNextData()
    delay 20 ms

getProcessTemp()  …
shutdownProcess() …
sendNextData()  …
```

Is often better rewritten as

```
global gTemp
loop():
  checkProcessTemp()
  checkForShutdown()
  handleComms()
  delay 20 ms
```

With the details in functions called by the superloop.

Note also, the use of a global variable, gTemp, to communicate between different task functions

```
checkProcessTemp():
  gTemp = getProcessTemp()
checkForShutdown ():
  if gTemp outside limits
    shutdownProcess()
handleComms():
  if commsLink ready AND data in queue
    sendNextData()
```

Contd.

- Decomposition into functions
  - Generally use a separate function for each individual IO task at least
  - Can also put simple tightly coupled activities into a single function
- Decide on the superloop timing
  - Base this on the task with fastest timing (shortest period) needs
  - For now, just use a delay to approximate timing simply (later we’ll look at timers)
- We’ll revisit this method in more detail later…

example

- Guidelines
  - Use clearly identified global variables/data to communicate information or state between tasks
  - Generally try to ensure only one function ever modifies the global variable
- Example
  - System with single LED
  - Enable/disable the LED using a button
  - When enabled, flash an LED on/off once every second

```
global gAppIsEnabled = FALSE
main():…
setup():
  set LED = OFF


loop():
    checkButtonPressed()
    updateLED()
    delay BUTTON_RESPONSE_TIME
checkButtonPressed():
  if debouncedButton has been clicked
    toggle gAppIsEnabled
updateLED():
  if gAppIsEnabled AND LED_FLASH_TIME has passed
    toggle LED
  else
    set LED = OFF
```

9

Contd.

- Guidelines
  - In some cases, you can put simple tightly coupled activities into a single function
  - But be careful: it can make your programme less flexible if you want to change later
- Example
  - LED1 is intially on, LED2 is initially off
  - Each time button1 pressed, toggle LED1 and LED2 immediately

```
main():…
setup():
  LED1 = on
  LED2 = off


loop():
    checkButton()
    delay BUTTON_RESPONSE_TIME


// button press is tightly
// coupled to toggling LEDs
// so put both in one function
checkButton():
  if button debounced/pressed
    toggle LED1
    toggle LED2
```

10

Embedded C differences

11

11

PIC16F87A Hardware Features & Constraints

12

- Native data word size is 8 bits
- Just 368 bytes of data RAM (for variables)
  - Uses segmented addressing (4 banks of 128 addresses)
  - Other addresses for special purpose registers and data mirroring/shadowing
- Instruction set limitations
  - No hardware multiply/divide
  - Limited indirection – affects pointers
- Just 8K program memory (for code)
  - Segmented addressing (4 pages of 2KiB each)
- Limited hardware stack (just 8 levels)
- Just one interrupt vector (software polled interrupts)

12

How does embedded C differ?

- Using the PIC C as the example…
- Provides access to unique features of hardware
  - Compiler options (e.g. --`CHIP=`*`processor`*)
  - special keywords, e.g.

```
void interrupt myInterruptHandler(void) { … }
volatile static unsigned char someVarSharedWithISR;
static bank3 unsigned char someVar;
```

  - Macros, e.g

```
bitset(var,bitnum)
__CONFIG(…)
EEPROM_WRITE(addr,value)
```

- Provides access to absolute addresses
  - E.g. to get/set values in special function registers

```
volatile unsigned char portA @ 0x05;
```

C differences contd.

- Data types
  - Native int and float size (see next slide)
  - Various limitations on pointers (see manual)
- Constraints and conventions for functions
  - Under the hood implementation (and performance) depends on number of arguments
  - Recursion (function calling itself) is supported but limited by the stack
    - Certain algorithms (common in algorithms books) cannot be used
  - Function nesting (f1 calls f2 calls f3 etc.) is also limited by stack

Contd (data type differences)

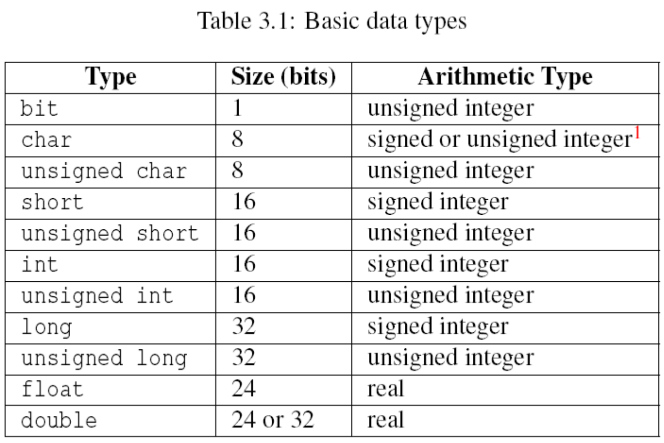
Table 3.1: Basic data types

Type	Size (bits)	Arithmetic Type
`bit`	1	unsigned integer
`char`	8	signed or unsigned integer[1]
`unsigned char`	8	unsigned integer
`short`	16	signed integer
`unsigned short`	16	unsigned integer
`int`	16	signed integer
`unsigned int`	16	unsigned integer
`long`	32	signed integer
`unsigned long`	32	unsigned integer
`float`	24	real
`double`	24 or 32	real

Table reproduced from HI-TECH PICC Compiler manual

Working with bits

- In embedded C you will often need to specify bit patterns or addresses
  - You should use binary (which is a Hitech C extension) or hex representation – octal is never needed
  - Binary is good if you want to be clear exactly which bits are high/low
- Bit patterns and addresses generally use unsigned integers

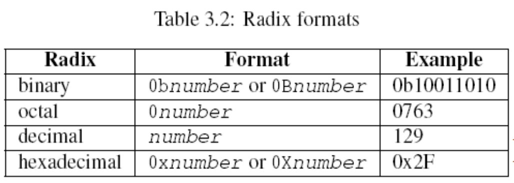
Table 3.2: Radix formats

Radix	Format	Example
binary	`0b`*number* or `0B`*number*	0b10011010
octal	`0`*number*	0763
decimal	*number*	129
hexadecimal	`0x`*number* or `0X`*number*	0x2F

Not standard C

Rarely useful these days

standard C (and useful)

Table reproduced from HI-TECH PICC Compiler manual

Working with bits contd.

- Bitwise operations and bit masks are much more common in embedded systems (and device driver firmware) than in normal desktop programmes, e.g. for manipulating configuration and status bits
- The main bitwise operators in C are:
  - **~a** (bitwise NOT), **a & b** (bitwise AND)
  - **a | b** (bitwise OR), **a ^ b** (bitwise XOR)
  - **a << b** (bitwise left shift by b places), **a >> b** (bitwise right shift by b places)
- Bitwise operators act on the individual bits in a number or corresponding bits in a pair of numbers.

17

bitwise shift in detail

- NOTE: When we write a binary number, MSb is leftmost digit, LSb is rightmost digit and is bit 0. The next bit to the left is bit 1 and so on.
- Shifting moves all bits in a number in one direction
  - One or more bits fall off the end (which end depends on direction)
  - One or more zeros get inserted to opposite end
  - E.g. 10**111111** left-shifted by 2 bits is **111111**00
- Left shift
  - Pseudocode: y = x left shifted by b bits
    C Code: y = x << b;
    Math equivalent: $y = x * 2^b$
  - E.g. y = x << 3 equivalent to y = x * 8
- Right shift
  - Pseudocode: y = x right shifted by b bits
    C code: y = x >> b;
    math equivalent: $y = x / 2^b$
  - E.g. y = x >> 1 equivalent to y = x / 2

Any thoughts on why shifting bits right or left like this could be useful?

18

Bitwise operation examples

Assume a = 0x7, b = 0x25, n = 3

- Bitwise not: **~a**
- Bitwise and: **a & b**
- Bitwise or: **a | b**
- Bitwise xor: **a ^ b**
- Bitwise left shift: **a << n**
- Bitwise right shift: **a >> n**

19

Self test

```
unsigned char x = 0x7;
unsigned char y;
unsigned char result;
unsigned char mask = 0b00110000;

y = x & mask;             // value of y?
             0x00  0b00110111  0x7  0
z = x | mask;             // value of z?

z = x & ~mask;            // value of z?

z = x ^ ~mask;            // value of z?
```

20

Bit masks (important)

- Bit masks are used to mask off (i.e. ignore and don't affect) some bits of a number

  See also http://en.wikipedia.org/wiki/Mask_(computing)

- A bitmask is a binary number that is used to help with certain bitwise operations
  - Its value identifies bits of interest (selected bits) and bits to be ignored (masked bits)
  - The bitmask is used to control which bits we access when performing some operation on a number. In the bitmask, 1's mark selected bits of interest and 0's mark bits to be masked off or ignored.
  - E.g. The bitmask 0b10010000 indicates that bits 7 and 4 are of interest while all others should be ignored.

Contd.

- Some key things you can do with a bitmask are:
  - Use the bitmask to select some bits of interest in a number or variable
  - Get the value of bits in the number selected by the bitmask and ignore any non-selected bits
  - Set (overwrite) the value of bits in the number selected by the bitmask and ensure any non-selected bits remain unmodified
  - Toggle the value of bits in the number selected by the bit mask
  - Clear the value of the bits in the number selected by the bit mask to zeros
  - Set the value of the bits in the number selected by the bit mast to ones

Bitmask worked example (PIC 16F877)

ADCON0 is an 8 bit register used for A/D conversion. Assume ADCON0 is initially 0b10110001. Bits 5,4,3 select the analog channel to be converted (where bit 0 refers to the LSb).

Q1. How many different channels can ADCON0 identify and what is the range of channel numbers (min…max)?

*Q2. if ADCON0 is 0b10**110**001, what channel is being selected?*

Q3. What bitmask should we use to access (either to get or set) the channel number while ignoring and not affecting other bits in ADCON0?

*Q4. How would you get the channel number from ADCON0 and assign it to the unsigned char variable **chan** using C code?*

*Q5. If the variable **newChan** was currently 4, how would you set this channel number in ADCON0 without affecting any other bits in ADCON0 using C code?*

Contd.

To get the currently selected channel…

```
chan = (ADCON0 & ADCHAN_MASK) >> 3;
```

```
var = (number & bitmask) >> numPlaces;
```

Example: assume ADCON0 is currently 0b10110001. Then break the statement above into multiple parts to see how it works

```
                                            ADCON0  = 0b10110001
chan = ADCON0 & ADCHAN_MASK;                chan =   (0b10110001
                                              BW-AND 0b00111000)
                                                 =    0b00110000
chan = chan >> 3;                           chan =    0b00000110  = 6
```

Contd.

To set the ADC channel select bits without affecting the value of other bits in ADCON0 (assuming the desired channel is specified in the variable newChan):

```
ADCON0 = (ADCON0 & ~ADCHAN_MASK) | (newChan << 3);
```

```
var = (number & ~bitmask) | (newValue << numPlaces);
```

Example: again assume ADCON0 is currently 0b10110001 and that the channel we want to select is channel 4. Breaking the statement above into its parts to see how it works we get…

```
                                      ADCON0 = 0b10110001
newChan = 4;                          newChan = 0b00000100 = 4
tmp1 = ADCON0 & ~ADCHAN_MASK;         tmp1 =    (0b10110001
                                         BW-AND 0b11000111)
                                            =    0b10000001
tmp2 = newChan << 3;                  tmp2 = 0b00100000
ADCON0 = tmp1 | tmp2;                 ADCON0 = (0b10000001
                                            OR   0b00100000
                                             =   0b10100001
```

Self test question

Assuming we have a binary number 0b0011 1101

Q1. How would you toggle all the bits in the number?

Q2. How exactly would you toggle selected bits (e.g. bits 2 and 4) in the number?

Hint: take a look at XOR

Self test questions

27

*Assume the variable **status** contains the value 0b10011011.*

Q1. Declare and initialize the bitmask to select bits 0,2,5 and 7

Q2. Write the code to clear the selected bits using the bitmask

Q3. Declare and initialize a second bitmask to select bits 3 and 4 which together represent the "mode"

*Q4. Write the code to get the "mode" and save it to the variable **oldMode***

Q5. Write the code to set the "mode" to the literal value 2

27

C coding – revision topics

28

28

C naming/style guide for labs

29

- We will try to use a consistent coding style in lectures and labs – you must do this with your code also
- Layout each C file with standard sections and extensive comments
- Make use of constants, #defines, and sensible function/variable naming to make code more readable
- **General names (including functions and variables)** are mixed case (lowercase initial, internal word caps), e.g. `myFunctionName`
- All enum, structure, and other user defined **types** are written as mixed case capitalised words, e.g. `MyType`
- **Constants and Macros** are all caps with "_" between words, e.g. `THIS_IS_A_CONSTANT`

29

Example C program:flash_leds.c

30

```
/** Flash LEDs [etc…] */
#include        <pic.h>
// configuration _____________
__CONFIG(WDTDIS&XT&UNPROTECT);
// types ______________________
typedef unsigned int  Uint;
typedef unsigned char Boolean;
// constants __________________
#define LED1 RB1
#define SUPERLOOP_DELAY 250
// globals ____________________
Boolean gIsBlinking = TRUE;
// macros _____________________
#define TOGGLE_BIT(b)   b ^= 1
// prototypes _________________
void updateLEDs(void);
void delay(Uint);
```

```
// top level functions =============
/** main entry point for system
 */
void main(void) {
  init();
  // superloop…
  for (;;) {
    updateLEDs();
    delay(SUPERLOOP_DELAY);
  }
/** blink LEDs */
void updateLEDs(void) {
  TOGGLE_BIT(LED1);
}
// helper functions ================
/** delay using a busy wait loop
 */
void delay(Uint nLoops) {
  for ( ; nLoops>0; nLoops--)
    asm("nop");
}
```

Note: the ordering of the recommended sections within the file and the comments that highlight the start of each section

30

Revision topics

- The following slides briefly list the main topics in C coding that you should be familiar with from your previous modules
- If your C programming is rusty, please review the content from modules such as EE108

Revision: functions

- Function **prototype** (AKA declaration)
  - Tells compiler (and us) a function's name and parameter types
  - Put this at top of file for single C file app, or in a separate include file for multi-C-file app
- Function **implementation** (AKA definition)
  - This is the actual code for the function
- Function **call** (AKA usage)
  - Every time you need to use/invoke/call the function
  - The parameters you pass into the function call must match the function prototype
  - Functions with a void return type, do not return values
  - Functions which return values
    - can be used on the right hand side of an assignment, e.g. `retVal = someFunction();`
    - Can be used wherever a value could be used, such as a parameter to another function or in an if statement, e.g. `y = square(square(x));`

examples

```
// PROTOTYPES___________
// NOTE: parameter types
// but no names

void delay(int);
char charToNumber(char);

// IMPLEMENTATIONS______

void delay(int nLoops) {
  // code goes here – not
  // shown for brevity…
}

char charToNumber(char c) {
  if ('0' < c) && (c < '9')
    return c – '0';
  else
    return -1;
}
```

```
int x = 4;
char c = '6';
char num;

// CALLS_______________

delay(x);
delay(27);
delay(x * 3);

delay(charToNumber('5'));
num = charToNumber(c);
delay(num);

if (num > 2)
  …
if (charToNumber(c) > 2)
  …
```

Revision: arrays and strings

- Array
  - A sequence (ordered collection) of data elements
  - Data elements can be primitive types (char, int, etc) or compound types (structures)
  - A string in C is stored in an array of characters
    - The end of string is indicated using the nul terminator ('\0')
    - The number of characters in the string cannot be more than the array size minus 1 (for the nul terminator), but can be less if you need/wish
- Pointers and arrays
  - Pointer to whole array
    - The name of the array is a pointer to the first element which is a pointer to array
  - Pointer to individual elements within array
    - Pointer arithmetic using the pointer to the array, or by taking the address of any array element (e.g. &myArray[4])
- Array initialization
  - Declare space used by array, and fill in values elsewhere
  - Or specify initial values when declaring array

Examples (ordinary arrays)

35

```
// initializing arrays
int array1[10]; // value??
int array2[] = {1,2,3};
// pointers
int* p1;
int* p2;
int* p3;
// what do these lines do?
p1 = array2;
p2 = &array2[0];
p1++;
*p1 = 20; // array contents?
p3 = p1 + 2;
*p3 = 30; // array contents?
p3 = &array[1];
*p3 = 40; // array contents?
```

35

Examples (strings)

36

```
// a char is just a number
char c1 = 27;
char c2 = 'A'; // use ASCII code of 'A'
c2 += 1; // now what's the value?


// initializing strings…
// strings are just arrays of characters
// **MUST** be null terminated however!
char str1[6] = {'a','b','c','d','e','\0'};
char str2[] = {'a','b','c', 0}; // note: 0 same as '\0'
char str3[] = "abcde"; // nul terminator implied by double quotes
// string pointers - what do these lines do??
char* p1 = str1;
char c3 = *(p1++);
c3 = *p1;
char* p2 = &str1[1];
```

36

Revision: Structures

37

- A structure (or record) is a compound data type
  - Used to keep related values together, e.g. measurement value and units
  - Parts of the structure are called **fields**
- Type must be declared before being used to declare a variable
- Intializing values
  - Like arrays, can initialize the whole structure at once at place of declaration
  - Or field by field later (e.g. in an initialization function)
- To refer to fields use
  - Structure name + dot notation (if variable is visible in current scope, e.g. local variable in current function)
  - Structure **pointer** + arrow notation

37

examples

38

```
// structure declaration - define the fields in this structure
struct RangeType {
  int min;
  int max;
};

// variable declaration - an instance of the structure
struct RangeType s1 = { 50, 100 };
struct RangeType s2; // value??

// dot notation when structure variable is in scope
s2.min = 10;
s2.max = 30;

// struct pointer
struct RangeType *ps = &s2; // where does ps point?

// array notation (and equivalent dot notation) when using a
// structure pointer, typically when structure variable is NOT
// in scope

ps->min = 20;
ps->max = 90;
(*ps).min = 20;
(*ps).max = 90;
```

38

Revision: constants

39

- It is very often necessary to use constants in a C program (typically numbers and strings)
  - Typically used for timing/delay values, threshold values, conversion factors, states that a system or device can be in, etc.
  - **Avoid literal values** (i.e. numbers or strings) in the body of your code – **use constants instead** (defined at top of file or elsewhere)
- Three ways to define constants
  - #define: easy for numbers or strings (but not type safe)
  - enum: an enumerated type is good for defining states
  - const: easy for numbers, strings, and compound data types, but a bit more difficult with pointers

39

examples

40

```
// declaring constants
#define MIN_TEMP   -20
#define HELLO_MSG  "Hello World"
enum CommsStateType {
    CS_READY,
    CS_WAITING,
    CS_BUSY
};
const int MAX_TEMP = 4;
// using constants
int x;
enum CommsStateType cs = CS_READY;
x = MIN_TEMP;
x = MAX_TEMP;
```

40

Typedef

- A typedef in C is used to define an alias (new name) for some existing type
  - Usually the alias is more convenient and readable than the existing name of the type
  - Commonly used with structures, enumerated types, and some numeric types
- Format is:

```
typedef oldname newname
```

examples

```
// declaring typedefs
typedef unsigned char   Byte;
typedef unsigned char   Boolean;

enum BtnStateType {
  BS_UP=1,
  BS_DOWN=0
};
typedef enum BtnStateType   BtnState;

typedef enum {
  LS_OFF=0,
  LS_ON=1
} LEDState;

struct BufferType {
  int used;
  int free;
};
typedef struct BufferType   Buffer;

typedef struct {
  int min;
  int max;
} Range;
```

```
// using typedef'd types
Byte x = 42;

LEDState lsVar = LS_OFF;

Range s1;
```

Revision: operators in C

- Arithmetic operators
  - =, +, -, *, /
  - % (modulus: **a % b** = remainder of a divided by b)
  - Be careful of wrap around when using signed values, e.g.

    ```
    char x = 120; x += 10; // what value is x?
    ```
  - Be careful of wrap around when using unsigned values, e.g.

    ```
    unsigned char x = 250; x += 10; // what value is x?
    ```
- Compound assignment operators
  - Any bitwise or arithmetic operator followed by equals
  - E.g. **a += b** is the same as **a = a + (b)**
  - E.g. **a |= b** is the same as **a = a | (b)**

43

Contd.

- Logical operators
  - Mainly used in the context of if-statements
  - Logical operators do not compare bit by bit, instead 0 is false and any non-zero value is true
  - **a && b** (Logical AND), **a || b** (Logical OR), **!a** (Logical NOT)
- Bitwise operators
  - Widely used for working with bit masks, configuration and status bits, etc.
  - Bitwise operators act on the individual bits in a number or pair of numbers
  - **~a** (bitwise NOT), **a & b** (bitwise AND)
  - **a | b** (bitwise OR), **a ^ b** (bitwise XOR)
  - **a << b** (bitwise left shift), **a >> b** (bitwise right shift)

44

examples

```
// why is this wasteful?

int trueOrFalse = 0;

// mod operator

char a = 2;
char b;
b = ++a % 4; // value of b?
b = ++a % 4; // value of b?

// wrap around
// assume char is signed
// by default

char a1 = 120;
char b1 = a1 + 12; // value?
unsigned char c1=0;
c1--;              // value?
```

```
// logical vs. bitwise
// operators

char a2 = 0x03;
char b2 = 0x07;
char c2;

c2 = a2 && b2; // value?
c2 = a2 & b2; // value?

// bit shifts

unsigned char x = 0x01;
x = x << 2;         // value?
x <<= 2;            // value?
x |= 0b00001111;   // value?
```

45