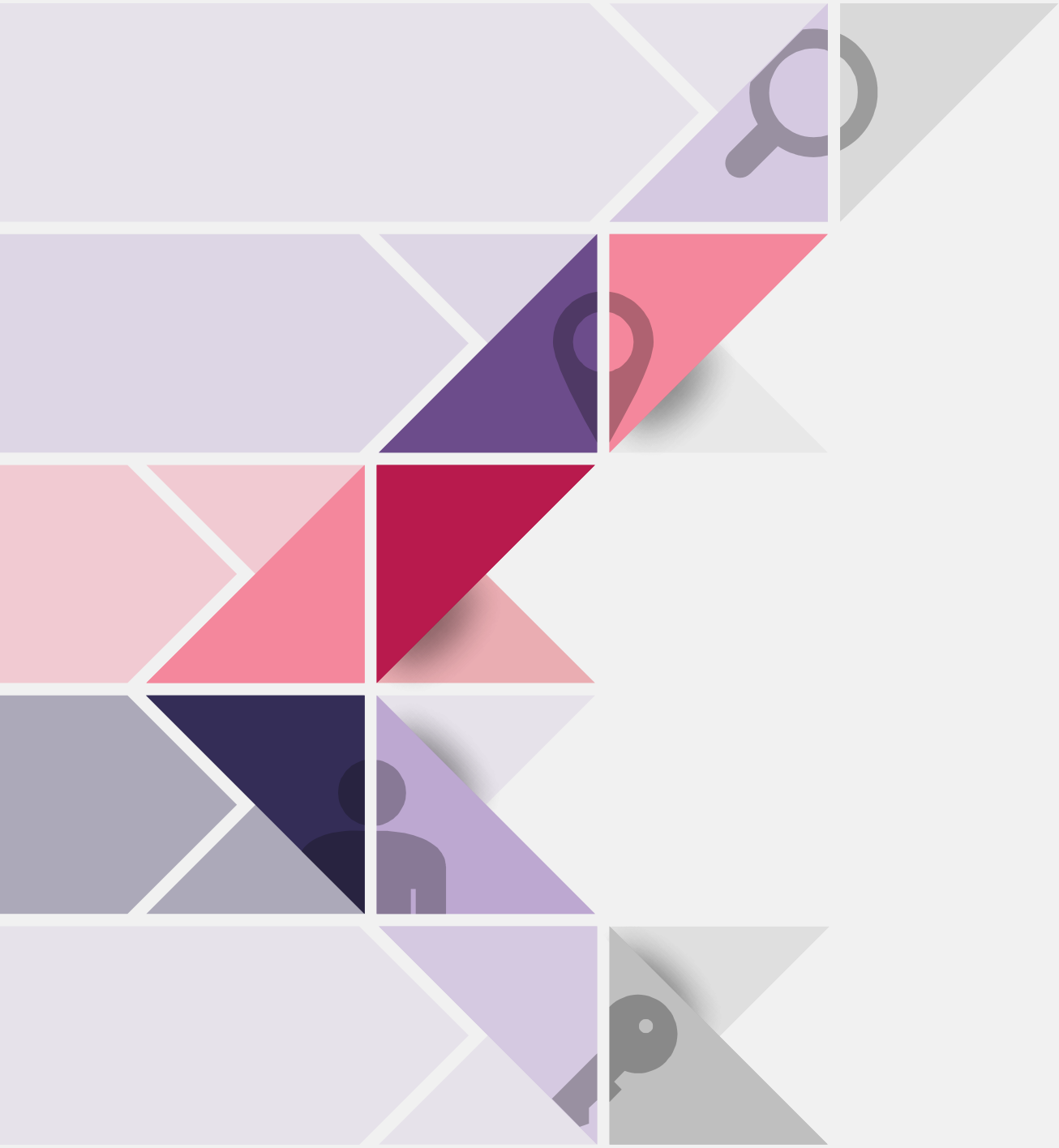
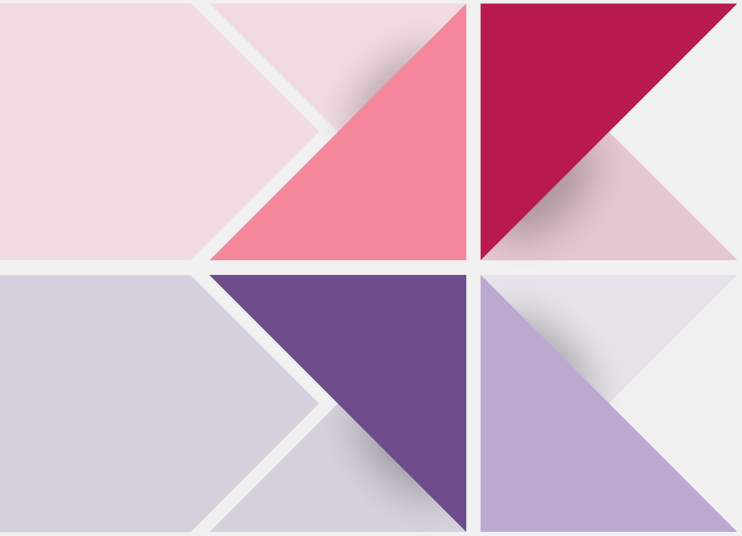


String

Index



01
String

String

String Literal

Strings are arrays of characters in which a special character - the null character - marks the end

A string literal is a sequence of characters enclosed within double quotes

"when you come to a fork in the road, take it."

String literals may contain escape sequences

Character escapes often appear in *printf* and *scanf* format strings

```
"Candy\nIs dandy\nBut liquor\nIs quicker.\n  --Ogden Nash\n"
```

```
Candy
Is dandy
But liquor
Is quicker.
   --Ogden Nash
```

String

String Literal

The backslash character (\) can be used to continue a string literal from one line to the next

```
printf("When you come to a fork in the road, take it.  \
    --Yogi Berra");
```

When two or more string literals are adjacent, the compiler will join them into a single string

This rule allows us to split a string literal over two or more lines

```
printf("When you come to a fork in the road, take it.  "
          "--Yogi Berra");
```

String

String Literal

When a C compiler encounters a string literal of length n in a program, it sets aside n + 1 bytes of memory for the string

This memory will contain the characters in the string, plus one extra character - the *null character* - to mark the end of the string

The null character is a byte whose bits are all zero, so it's represented by the \0 escape sequence

The string literal "abc" is stored as an array of four characters

The string "" is stored as a single null character

String

String Literal

Since a string literal is stored as an array, the compiler treats it as a pointer of type char *

Both printf and scanf expect a value of type char * as their first argument

```
int printf ( const char * format, ... ); int scanf ( const char * format, ... );
```

The following call of printf passes the address of "abc" (a pointer to where the letter a is stored in memory)

```
printf("abc");
```

String

String Literal

We can use a string literal wherever C allows a char * pointer

```
char *p;
p = "abc";
```

String literals can be subscripted as following, the new value of ch will be the letter b

```
char ch;
ch = "abc"[1];
```

```
char *p = "abc";
*p = 'd';
```

A function that converts a number between 0 and 15 into the equivalent hex digit

```
char digit_to_hex_char(int digit)
{
    return "0123456789ABCDEF"[digit];
}
```

String

String Literal

A string literal containing a single character isn't the same as a character constant

"a" is represented by a *pointer*
'a' is represented by an *integer*

```
prinft("\n");

prinft('\n');     //Error
```

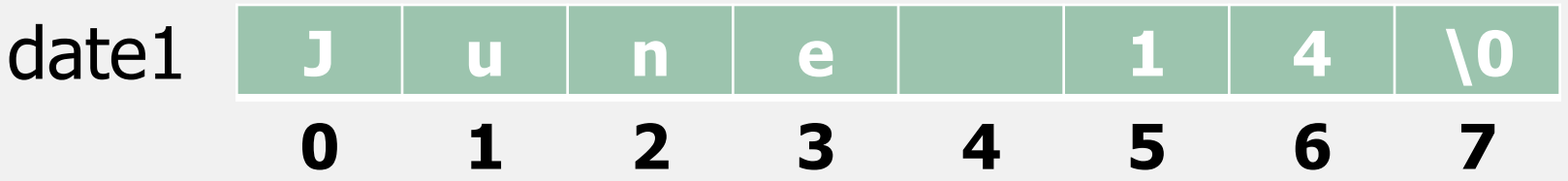
String

String Initialization

If a string variable needs to hold 80 characters, it must be declared with length 81 because of the end of string '\0'

```
char date1[8] = "June 14";
```

The compiler will automatically add a null character



"June 14" is not a string literal in this context

String

String Initialization

If the initializer is too short to fill the string variable, the compiler will insert extra null characters

```
char date2[9] = "June 14";
```

Then the date2

date2	J	u	n	e		1	4	\0	\0
	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**

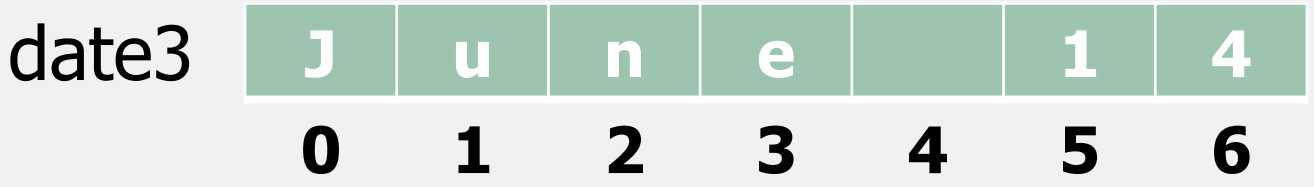
String

String Initialization

An initializer for a string variable can't be longer than the variable, but it can be the same length

```
char date3[7] = "June 14";
```

Then the date3




```c
#include <stdio.h>

int main()
{
    char date3[7] = "June 14";
    printf("%s\n", date3);

    return 0;
}
```

```
June 14   a
```

String

String Initialization

The declaration of a string variable may omit its length, in which case the compiler computes it

```
char date4[] = "June 14";
```

Then the compiler sets aside eight characters for date4, enough to store the characters in "June 14" plus a null character

String

String Initialization

The declaration as following declares date to be an array

```
char date[] = "June 14";
```

The similar-looking declares date to be a pointer

```
char *date = "June 14";
```

However, there are significant differences between the two date

- In array version
  - The characters stored in date can be modified
  - The data is an array name
- In pointer version
  - The date points to a string literal that shouldn't be modified
  - The data is a variable that can point to other strings

String

String Initialization

Using an uninitialized pointer variable as a string is a serious error

An attempt at building the string "abc"

```
char *p;
p[0] = 'a';
p[1] = 'b';
p[2] = 'c';
p[3] = '\0';
```

Because p hasn't been initialized, it causes undefined behavior

String

String Library

To print part of a string, use the conversion specification %.ps

The statement is

```
char str[] = "Are we having fun yet?";
printf("%.6s\n", str);
```

Output is

```
Are we
```

The C library also provides puts function

```
puts(str);
```

After writing a string, puts always writes an additional new-line character

String

String Library

The %s conversion specification allows scanf to read a string into a character array

```
scanf("%s", str);
```

str is treated as a pointer, so there’s no need to put the & operator in front of str

When scanf is called, it skips white space, then reads characters and stores them in str until it encounters a white-space character

scanf always stores a null character at the end of the string

String

String Library

Consider the following program fragment

```
char sentence[SENT_LEN+1];

printf("Enter a sentence:\n");
scanf("%s", sentence);
```

If the input is

To C, or not to C: that is the question.

scanf will only store the string "To" in sentence

String

String Library

A new-line character will cause scanf to stop reading, but so will a space or tab character

To read an entire line of input, gets can be used

- Doesn't skip white space before starting to read input
- Reads until it finds a new-line character
- Discards the new-line character instead of storing it; the null character takes its place

```
gets(sentence);
```

To C, or not to C: that is the question.

String

String Library

As they read characters into an array, scanf and gets have no way to detect when it’s full

Consequently, they may store characters past the end of the array, causing undefined behavior

scanf can be made safer by using the conversion specification %ns instead of %s

gets is inherently unsafe; fgets is a much better alternative

String

String Library

A program to read a line using getchar() function

```
int read_line(char str[], int n)
{
    int ch, i = 0;

    while ((ch = getchar()) != '\n')
        if (i < n)
            str[i++] = ch;
    str[i] = '\0';      // Terminates string
    return i;           // Number of characters stored
}
```

ch has int type rather than char type because *getchar*() returns an int value

String

String Library

A function that counts the number of spaces in a string

```
int count_spaces(const char s[])
{
      int count = 0, i;

      for (i = 0; s[i] != '\0'; i++)
          if (s[i] == ' ')
            count++;
      return count;
}
```

String

String Library

A version that employs pointer arithmetic instead of array subscripting

```
int count_spaces(const char *s)
{
    int count = 0;

    for (; *s != '\0'; s++)
        if (*s == ' ')
            count++;
    return count;
}
```

String

String Library

Questions in the count_spaces function

- Q1: Is it better to use array operations or pointer operations to access the characters in a string?
- Ans: We can use either or both
- Q2: Should a string parameter be declared as an array or as a pointer?
- Ans: There's no difference between the two
- Q3: Does the form of the parameter (s[] or *s) affect what can be supplied as an argument?
- Ans: No

String

String Library

Direct attempts to copy or compare strings will fail

Copying a string into a character array using the = operator is not possible

```
int main()
{
    char str1[10], str2[10];
    str1 = "abc";
    str1 = str2;

    return 0;
}
```

```
test.c:7:10: error: assignment to expression with array type
     str1 = "abc";
          ^
test.c:8:10: error: assignment to expression with array type
     str1 = str2;
          ^
```

Using an array name as the left operand of = is illegal

Initializing a character array using = is legal

```
char str1[10] = "abc";
```

String

String Library

Attempting to compare strings using a relational or equality operator is legal but won't produce the desire result

```
if (str1 == str2) ...   //Error
```

Why? Because this statement is the pointer comparison

The C library provides a rich set of functions for performing operations on strings

- strcpy and strncpy

```
#include <string.h>
char *strcpy(char *s1, const char *s2)                  //Copy string s2 to s1
char *strncpy(char *s1, const char *s2, size_t count)   //Copy string s2 to s1 with  length count
```

String

String Library

Hence, if the length of str2 is greater than or equal to that of str1, the strncpy will leave str1 without a terminating null character

The safer way to use strncpy is

```
strncpy(str1, str2, (length of str1) - 1) ;
str1[(length of str1) - 1] = '\0';
```

The second statement guarantees that str1 is always null-terminated

String

String Library

- strlen
  - The function will return the string length with the unsigned integer type
  - The Prototype is

```
size_t strlen(const char *s) ;
```

- *size_t* is a *typeof* name that is one of C's unsigned integer types
- *strlen* returns the length of a string s, not including the null character

```
int len;

len = strlen("abc");        // len is now 3
len = strlen("");           // len is now 0
strcpy(str1, "abc");
len = strlen(str1);         // len is now 3
```

String

String Library

- strcat
  - The function will return a string which is the combination of two strings
  - The Prototype is

```
char *strcat(char *s1, const char *s2);
```

- *strcat* returns the string combination to s1 (a pointer to the resulting string)

```
strcpy(str1, "abc");
strcat(str1, "def");  // str1 now contains "abcdef"
strcpy(str1, "abc");
strcpy(str2, "def");
strcat(str1, str2);    // str1 now contains "abcdef"
```

```
strcpy(str1, "abc");
strcpy(str2, "def");
strcat(str1, strcat(str2, "ghi"));
/* str1 now contains "abcdefghi";
    str2 contains "defghi" */
```

String

String Library

- strcat
  - The function will return a string which is the combination of two strings
  - The Prototype is

```
char *strcat(char *s1, const char *s2);
```

- *strcat* returns the string combination to s1 (a pointer to the resulting string)

```
strcpy(str1, "abc");
strcat(str1, "def");  // str1 now contains "abcdef"
strcpy(str1, "abc");
strcpy(str2, "def");
strcat(str1, str2);    // str1 now contains "abcdef"
```

```
strcpy(str1, "abc");
strcpy(str2, "def");
strcat(str1, strcat(str2, "ghi"));
/* str1 now contains "abcdefghi";
    str2 contains "defghi" */
```

- *strcat*(str1, str2) might cause undefined behavior if the str1 array isn't long enough to accommodate the characters from str2

String

String Library

- strcmp
  - The function is a comparison function between two strings
  - The Prototype is

```
int strncmp(const char *s1, const char * s2);
```

- *strcmp* compares the string s1 and s2, returning a value less than, equal to, or greater than 0, depending on whether s1 is less than, equal to, or greater than s2

```
if (strcmp(str1, str2) < 0)          // is str1 < str2?
    …
```

String

String Library

- strcmp considers s1 to be less than s2 if either one of the following conditions is satisfied
  - The first i characters of s1 and s2 match, but the (i+1)st character of s1 is less than the (i+1)st character of s2
  - All characters of s1 match s2, but s1 shorter than s2

- As it compares two strings, strcmp looks at the numerical codes for the characters in the strings
  - A-Z, a-z, and 0-9 have consecutive codes
  - All upper-case letters are less than all lower-case letters
  - Digits are less than letters
  - Space are less than all printing characters

String

String Library

Write a program to print a One-Month Reminder List

```
Enter day and reminder: 24 Susan's birthday
Enter day and reminder: 5 6:00 - Dinner with Marge and Russ
Enter day and reminder: 26 Movie - "Chinatown"
Enter day and reminder: 7 10:30 - Dental appointment
Enter day and reminder: 12 Movie - "Dazed and Confused"
Enter day and reminder: 5 Saturday class
Enter day and reminder: 12 Saturday class
Enter day and reminder: 0

Day Reminder

  5 Saturday class
  5 6:00 - Dinner with Marge and Russ
  7 10:30 - Dental appointment
 12 Saturday class
 12 Movie - "Dazed and Confused"
 26 Movie - "Chinatown"
```

String

String Array

There is more than one way to store an array of strings

One option is to use a two-dimensional array of characters, with one string per row

```
char planets[][8] = {"Mercury", "Venus", "Earth",
                     "Mars", "Jupiter", "Saturn",
                     "Uranus", "Neptune", "Pluto"};
```

The number of rows in the array can be omitted, but we must specify the number of columns

String

String Array

0	1	2	3	4	5	6	7
M	e	r	c	u	r	y	\0
V	e	n	u	s	\0	\0	\0
E	a	r	t	h	\0	\0	\0
M	a	r	s	\0	\0	\0	\0
J	u	p	i	t	e	r	\0
S	a	t	u	r	n	\0	\0
U	r	a	n	u	s	\0	\0
N	e	p	t	u	n	e	\0
P	l	u	t	o	\0	\0	\0

String

String Array

Most collections of strings will have a mixture of long strings and short strings

Hence, a ragged array is needed whose rows can have different lengths

A ragged array can be created by using pointers to strings

```
char *planets[] = {"Mercury", "Venus", "Earth",
                   "Mars", "Jupiter", "Saturn",
                   "Uranus", "Neptune", "Pluto"};

for (i = 0; i < 9; i++)
    if (planets[i][0] == 'M')
      printf("%s begins with M\n", planets[i]);
```

M	e	r	c	u	r	y	\0
V	e	n	u	s	\0		
E	a	r	t	h	\0		
M	a	r	s	\0			
J	u	p	i	t	e	r	\0
S	a	t	u	r	n	\0	
U	r	a	n	u	s	\0	
N	e	p	t	u	n	e	\0
P	l	u	t	o	\0		

String

Command-line Argument

Examples of UNIX ls command

```
ls
ls -l
ls -l remind.c
```

Command-line information is available to all programs, not just operating system commands

To obtain access to command-line arguments in main

```
int main(int argc, char *argv[])
{
    ...
}
```

String

Command-line Argument

```
int main(int argc, char *argv[])
{
    ...
}
```

argc ("argument count") is the number of commend-line arguments

argv ("argument vector") is an array of pointers to the command-line arguments (stored as strings)

argv[0] points to the program name while argv[1] to argv[argc-1] point to the remaining command-line arguments

argv[argc] is always a null pointer

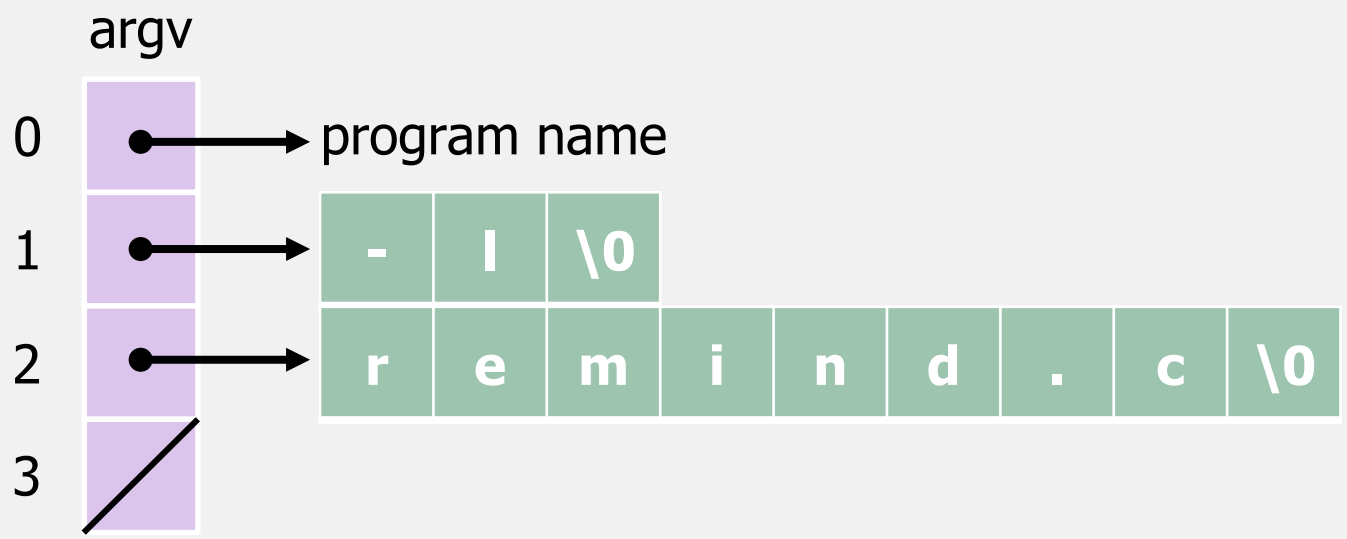
String

Command-line Argument

If the user enter the command line

ls -l remind.c

then the argc will be 3, and argv will be as the following




```
char **p;
for (p = &argv[1]; *p != NULL; p++)
      printf("%s\n", *p);
```

String

Command-line Argument

Write a program to check planet names using command-line arguments

Enter the command line

```
planet Jupiter venus Earth fred
```

Output

```
Jupiter is planet 5
venus is not a planet
Earth is planet 3
fred is not a planet
```

```
char *planets[] = {"Mercury", "Venus", "Earth",
                   "Mars", "Jupiter", "Saturn",
                   "Uranus", "Neptune", "Pluto"};
```