HIGH-LEVEL PROGRAMMING I

Arrays of strings

by Prasanna Ghali

Arrays of strings (1/3)

- Array of strings can be represented by two-dimensional array of characters with one string per row

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

Arrays of strings (2/3)


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

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

Arrays of strings (3/3)

- Unfortunately, `planets` array contains fair bit of wasted space (extra null characters)

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

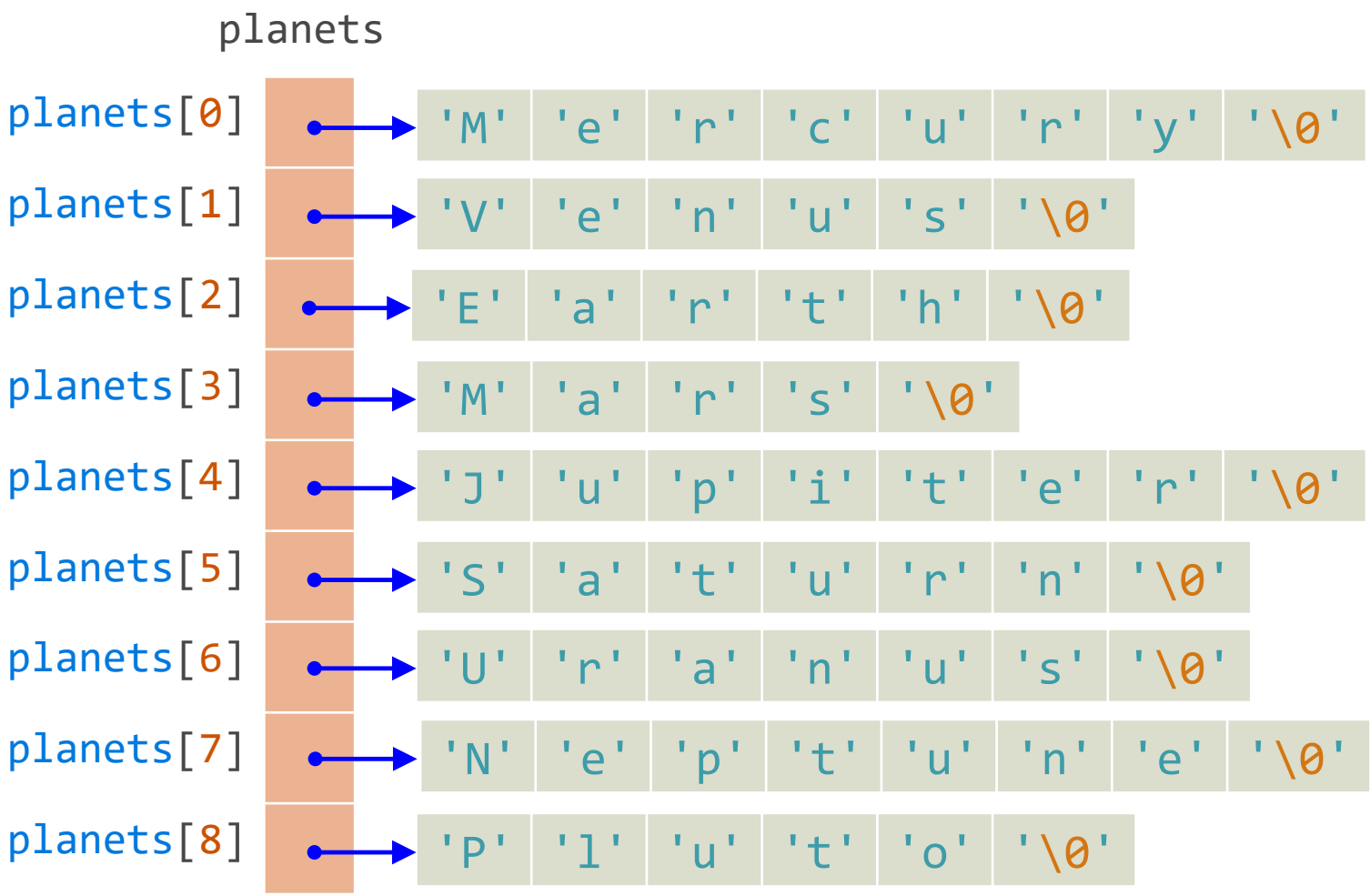
Ragged arrays (1/3)

- What is required is *ragged array* whose rows can have different lengths
- Ragged array can be simulated by creating an array whose elements are *pointers* to strings:

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

Ragged arrays (2/3)


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



Ragged arrays (3/3)

- Loop that searches array `planets` for strings beginning with letter M:

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

int const N = sizeof(planets)/sizeof(planets[0]);
for (int i = 0; i < N; ++i) {
  if (planets[i][0] == 'M') {
    printf("%s begins with M\n");
  }
}
```

Command-line parameters (1/4)

- When a program is run, it often must be supplied with information
  - May include file name(s) or switches that modify program's behavior
  - This information is called *command-line parameters*
- Example is copy command
  - `cp src-file-name dest-file-name`
- Another example is list directory command
  - `ls` or `ls –a` or `ls -l`

Command-line parameters (2/4)

- To access *command-line parameters*, `main` must have two parameters:

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

- Command-line parameters are called *program parameters* in the C standard.

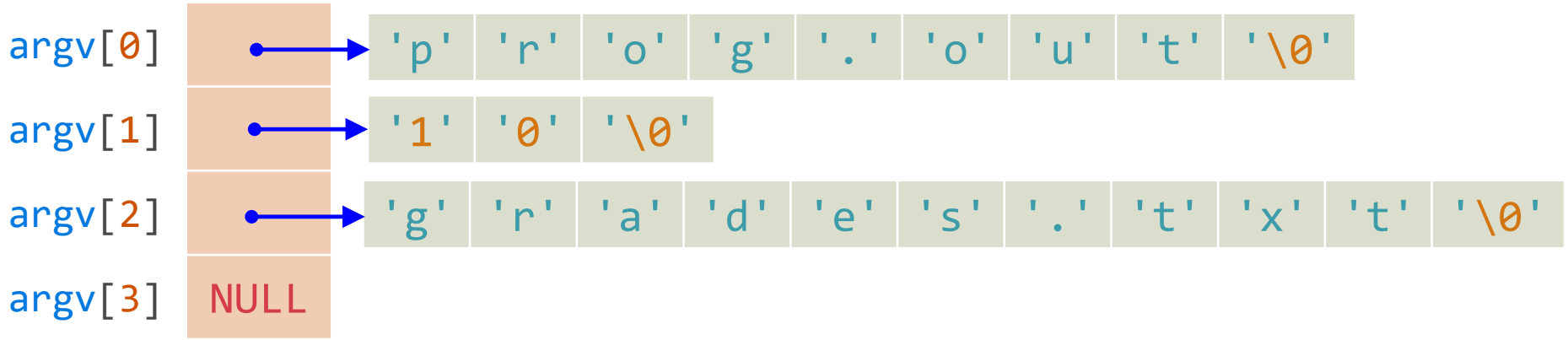
Command-line parameters (3/4)

- `argc` is count of command-line parameters
- `argv` is array of pointers to command-line parameters stored as strings
  - `argv[0]` points to program's name
  - `argv[1]` thro' `argv[argc-1]` point to remaining command-line parameters
  - `argv[argc]` always contains value `NULL` – that is, it is a *null pointer* that points to nothing

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

Command-line parameters (4/4)

- Assume user executes a program in this manner: *prog.out 10 grades.txt*
- `argc` equivalent to 3
- `argv` has type `char *argv[]` with form:



Processing command-line parameters (1/2)

- Iterate over elements in array `argv` using `int` variable as index

```
int main(int argc, char *argv[]) {
  // print command-line parameters
  for (int i = 0; i < argc; ++i) {
    printf("%s\n", argv[i]);
  }
  // other code ...
}
```

Processing command-line parameters (2/2)

- Iterate over elements in `argv` array using variable of type `char**` that initially points to 1st array element

```
int main(int argc, char **argv) {
  // print command-line parameters
  for (char **p = argv; *p; ++p) {
    printf("%s\n", *p);
  }
  return 0;
}
```

Program: *planets.c*

- Write program to check if command-line parameters are names of planets …