COMPLEX DECLARATIONS

Plan for Today

- Command-line Parameters
- Multidimensional Arrays
- Complex Declarations

Complex Declarations

- Pointers to functions
- Arrays of pointers
- Multidimensional arrays and pointers to arrays
- Scrambling complex declarations

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[]) {
   ...
}
```

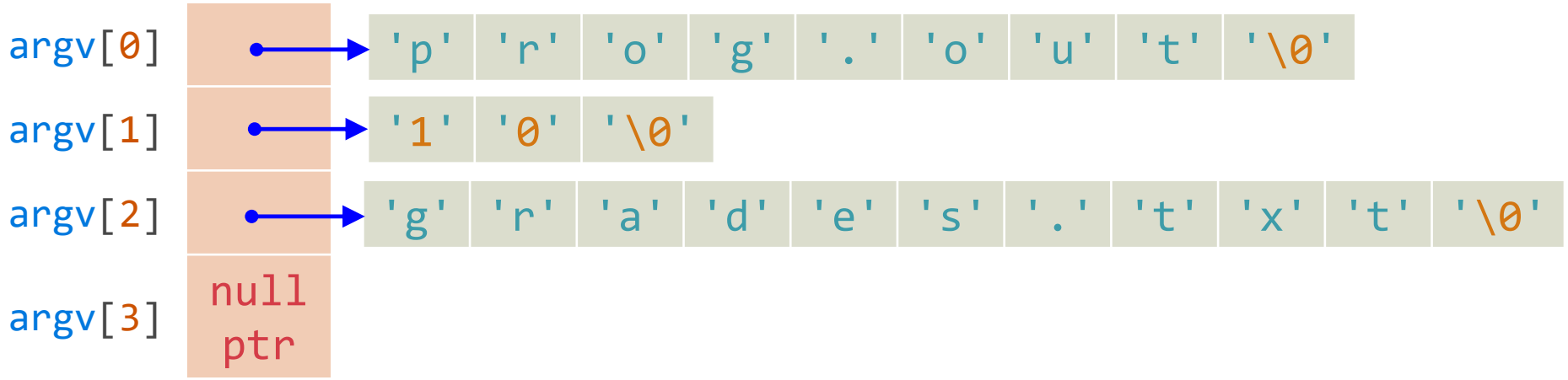
- Command-line parameters are called *program parameters* in the C standard.

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 a *null pointer* that points to nothing

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{}; i < argc; ++i) {
    std::cout << argv[i] << "\n";
  }
  // other code ...
}
```

Multi-Dimensional Arrays

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) {
    std::cout << *p << "\n";
  }
}
```

How a Declaration is Formed

- C declaration's come in two parts:
  - *base type* consists of type-specifier, storage-class specifier, and type-qualifier
  - *declarator* containing the identifier, or name being declared with the characters *, [], and () and possibly type-qualifiers

Declarator in C

How many	Name in C	How it looks in C

zero or more	pointers	one of the following alternatives:
		`* const volatile`
		`* volatile`
		`*`
		`* const`
		`* volatile const`

exactly one	direct_declarator	*identifer* or *identifer*`[`*optional_size*`]` … or *identifer* `(`*args...*`)` or `(`*declarator*`)`
zero or one	initializer	*assigment op initial_value*

C/C++ Declarations

How many	Name in C	How it looks in C
at least one type-specifier	type-specifier	`void char short int long signed unsigned float double` *`struct_specifier`* *`enum_specifier`* *`union_specifier`*
	storage-class	`extern static register auto typedef`
(not all combinations are valid)	type-qualifier	`const volatile`
exactly one	declarator	*see previous definition*
zero or more	more declarators	`, declarator`
one	semi-colon	`;`

Restrictions on Legal Declarations

- Not possible to have any of these:
  - function can't return a function, so `peach()()` never arises
  - function can't return an array, so `apple()[]` never arises
  - array can't hold a function, so `orange[]()` never arises

What's Allowed

- You can write any of these declarations:
  - `int (* grape())();`
  - `int (* pear()) []`
  - `int (* mango[]) ()`
  - `int kiwi[][]`

Precedence Rule

A. Declarations are read by starting with the name and then reading in precedence order
B. Precedence, from high to low, is:
    1. Parentheses grouping together parts of a declaration
    2. Postfix operators:
        Parentheses ( ) indicating a function, and
        Square brackets [ ] indicating an array
    3. Prefix operator: * operator denoting "pointer to"
C. If `const` and/or `volatile` keyword is next to type specifier (`int`, `long`, etc.) it applies to type specifier. Otherwise, `const` and/or `volatile` keyword applies to pointer asterisk on its immediate left

Precedence Rule: Example

- `char* const *(*next)();`
- `char *(*c[10])(int **p);`

Unscrambling Declarations by Diagram (1/2)

- Declarations in C are read boustrophedonically, i.e. alternating right-to-left with left-to-right
- Start at first identifier you find when reading from the left
- When a token in declaration is matched against diagram, erase it from further consideration
- At each point, look first to token to right, then to the left
- When everything has been erased, the job is done

Unscrambling Declarations by Diagram (2/2)