KYOTO UNIVERSITY OF ADVANCED SCIENCE

京都発世界人財

京都先端科学大学

Information Processing 2 10

"two-star" programming: pointers to pointers, arrays of pointers, command-line arguments, multi-dimensional arrays, pointers to functions

Ian Piumarta

Faculty of Engineering, KUAS

last week: memory, addresses, arrays and pointers

memory, addresses, variables, pointers, indirection
`swap(a, b)` that actually works
`getint()`
equivalence of arrays, elements, and pointers
`strlen()` using pointers
passing sub-arrays as arguments
negative indices; address arithmetic
`alloc()` and `afree()`
char array vs. char pointers vs. strings
versions of `strcpy()`, `strcmp()`

Chapter 5. Pointers and Arrays
5.1 Pointers and Addresses
5.2 Pointers and Function Arguments
5.3 Pointers and Arrays
5.4 Address Arithmetic
5.5 Character Pointers and Functions

this week: more arrays and pointers

arrays of strings
sorting lines of text from the input
modified `qsort()` that sorts strings
`day_of_year()` and `month_day()`
`month_name()` and initialising array of strings
`echo` program
find command line string in text lines
`sort` using function pointers
  to parameterise behaviour
many complex declarations
`dcl` converts declarations to English
`undcl` converts English to declaration

5.6 Pointer Arrays; Pointers to Pointers
5.7 Multi-dimensional Arrays
5.8 Initialisation of Pointer Arrays
5.9 Pointers vs. Multi-dimensional Arrays
5.10 Command-line Arguments
5.11 Pointers to Functions
5.12 Complicated Declarations

review: arrays and pointers

an array is a sequence of elements in memory

- all elements have the same type
- individual elements are identified by their *index* using ‘*array* `[]`’
- an index is an offset from the start of the sequence

when you pass an array as a function argument, you pass its *address* in memory

- this is why you can modify the contents and the caller ‘sees’ your modifications

```
void swapElements(int array[], int i, int j)
{
  int tmp = array[i];
  array[i] = array[j];
  array[j] = tmp;
}
```

review: arrays and pointers

a *pointer* is the address of a sequence of elements in memory

- there are zero or more elements at the pointer's address
- the element at an address is accessed by indirection using '**address*'
- pointers to adjacent elements are obtained by adding an offset to a pointer
- the offset is scaled by the element size, exactly like an array index

```
void swapIntegers(int *address, int i, int j)
{
  int tmp = *(address + i);           // tmp = addr[i];
  *(address + i) = *(address + j);    // addr[i] = addr[j];
  *(address + j) = tmp;               // addr[j] = tmp;
}
```

the correspondence between arrays and pointers is so strong that

$$\texttt{array[index]} \equiv \texttt{*(array + index)} \equiv \texttt{*(index + array)} \equiv \texttt{index[array]}$$

pointer arrays; pointers to pointers

pointers can be stored in variables (just like integers, floats)

pointers can therefore also be the elements of an array

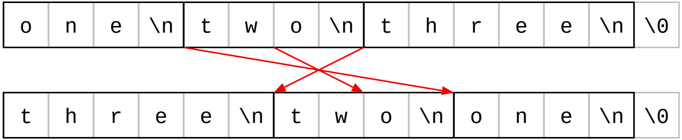
this enables efficient processing of variable-length data such as strings

reversing an array of integers is easy

- we already wrote the function

reversing lines of text in a long string is difficult

- you have to move all the characters around

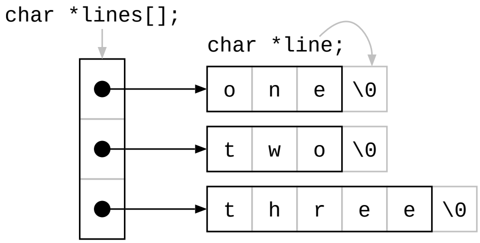


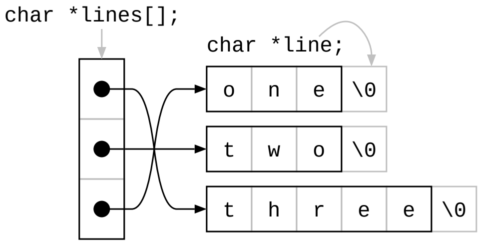
pointer arrays; pointers to pointers

the problem: given a sequence of *lines* of text, reverse the order of the *lines*

an array of *lines* of text would be a much better model

- ⇒ place each line into its own string (array of `char`)
- ⇒ make an array of pointers to those strings (array of `char *`)
- ⇒ reverse the order of the *pointers* to the *lines* in the array
  (do *not* move any of the individual characters in the strings themselves)





pointer arrays; pointers to pointers

reverse integers

```
void revints(int array[], int n)
{
  int l = 0, r = n - 1;

  while (l < r) {
    int tmp = array[l];
    array[l] = array[r];
    array[r] = tmp;
    ++l, --r;
  }
}
```

reverse strings

```
void revstrs(char *array[], int n)
{
  int l = 0, r = n - 1;

  while (l < r) {
    char *tmp = array[l];
    array[l] = array[r];
    array[r] = tmp;
    ++l, --r;
  }
}
```

pointer arrays; pointers to pointers

test program

```
int main()
{
  char *a = "one";
  char *b = "two";
  char *c = "three";

  char *lines[] = { a, b, c };

  printf("%s\n%s\n%s\n", lines[0], lines[1], lines[2]);

  revstrs(lines, 3);

  printf("%s\n%s\n%s\n", lines[0], lines[1], lines[2]);

  return 0;
}
```

pointer arrays; pointers to pointers

pointers (to elements *in the same array*) can be compared

instead of array indexes, we can use pointers to the elements being swapped

```
void revints(int *array, int size)
{
  int *left  = array;              // leftmost int in array
  int *right = array + size - 1; // rightmost int in array

  while (left < right) {   // more elements to swap
    int tmp = *left;       // remember left element
    *left++ = *right;      // replace it with right element
    *right-- = tmp;        // replace right element with original left
  }
}
```

pointer arrays; pointers to pointers

the same thing, using an array of ‘`char *`’ instead of ‘`int`’

```
void revstrs(char **array, int n)
{
  char **l = array;              // leftmost line in array to swap
  char **r = array + n - 1;      // rightmost line in array

  while (l < r) {        // pointers have not yet 'met' in the middle
    char *tmp = *l;      // swap *l with *r, at the same time...
    *l++ = *r;           //   move l one line to the right
    *r-- = tmp;          //   move r one line to the left
  }
}
```

if this is at all confusing, remember that: `char **array` ≡ `char *(*array)`
≡ `char *(array[])`
≡ `char *array[]`

multi-dimensional arrays

to make a two-dimensional array, declare an array of arrays

```
int a[3];      // a is an array of 3 ints
```

a:	a[0]	a[1]	a[2]

```
int b[2][3]; // b is an array of 2 arrays of 3 ints
```

b:	b[0][0]	b[0][1]	b[0][2]	b[1][0]	b[1][1]	b[1][2]

a two-dimensional array is really a one-dimensional array

- in which each element is another array

multi-dimensional arrays

a two-dimensional array is initialised by a list of elements in braces

- since each element is an array, it should also be a list of elements in braces

```
int powers[4][10] = {
  {  1, 1, 1,  1,  1,   1,   1,   1,   1,   1 },
  {  0, 1, 2,  3,  4,   5,   6,   7,   8,   9 },
  {  0, 1, 4,  9, 16,  25,  36,  49,  64,  81 },
  {  0, 1, 8, 27, 64, 125, 216, 343, 512, 729 },
};
```

```
  printf("7 cubed is %d\n", powers[3][7]);
```

note that `powers` contains storage for $4 \times 10 = 40$ `int`s whereas

```
int *other[4];
```

only contains storage for 4 pointers

multi-dimensional arrays

beware: `powers[3][7]` is not the same as `powers[3,7]`

to pass a two-dimensional array to a function, the parameter should be declared as

```
void f(int powers[4][10]) { ... }
```

or omitting the outermost dimension (as we did previously with strings)

```
void f(int powers[][10]) { ... }
```

or by declaring it a pointer to the element type (as we did previously with strings)

```
void f(int (*powers)[10]) { ... }
```

note that

- only the first (outermost) dimension can be omitted
  - the compiler needs to know the size of the elements of the array
- the parentheses in `int (*powers)[10]` are necessary
  - `int *powers[10]` is an array of 10 pointers to `int`
- arrays of pointers are more common than multi-dimensional arrays

initialisation of pointer arrays

pointer arrays are initialised with a list of elements in braces

each element must have the pointer type expected

```
int  a, b, c, *d[3] = { &a, &b, &c };
char p[3], q[4], r[5], *s[3] = { p, q, r };
```

note that each element of `s` is a pointer to an array of a different size
(this cannot be expressed as a two-dimensional array)

string constants have type `char *` and can be elements in an array of `char` pointers

```
char *t[3] = { "one", "two", "three" };
```

initialisation of pointer arrays

arrays often hold ‘look-up’ tables of constant data used internally by a function

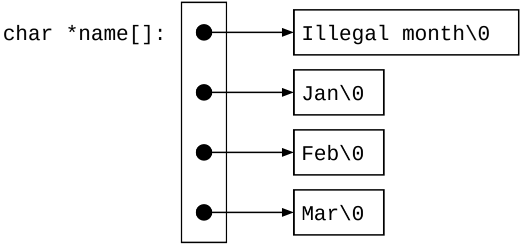
- use a `static` array to initialise the contents at compile time
- the array is a global variable but the name is local to the function

```
char *monthName(int n)
{
  static char *name[] = {
    "Illegal month",
    "January",   "February", "March",    "April",
    "May",       "June",     "July",     "August",
    "September", "October",  "November", "December",
  };
  return 1 <= n && n <= 12 ? name[n] : name[0];
}
```

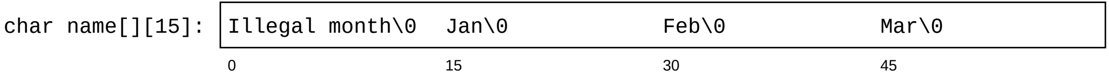
pointers vs. multi-dimensional arrays

we can declare `monthName` two ways

```
char *name[] = { "Illegal month", "Jan", "Feb", "Mar" };
```




```
char  name[][15] = { "Illegal month", "Jan", "Feb", "Mar" };
```



the former takes less space; the latter allows you to modify the names at run time

command-line arguments

when you run a program you can supply arguments to it on the command line

consider: `cc prog.c -o prog`

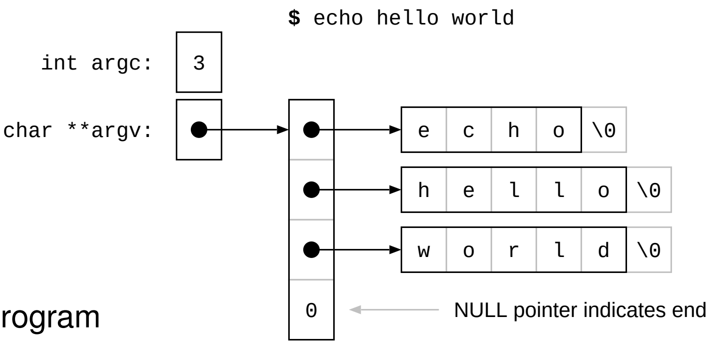
- ‘`cc`’ is the name of the program
- ‘`prog.c`’, ‘`-o`’, and ‘`prog`’ are additional arguments
- `cc` uses the arguments to receive essential information and modify its behaviour
  - the name of the source file to be compiled
  - don’t use the default output name
  - the name of the output file name that should be used

the `main` function receives these arguments as an array of strings

two arguments describe this array; the corresponding parameters are:

`int argc`	**arg**ument **c**ount: number of elements in `argv`
`char *argv[]`	**arg**ument **v**ector: array of strings containing the arguments

command-line arguments



`argv[0]` contains the name of the program

- `argc` is therefore always at least `1`
- if `argc == 1` then there are no command line arguments

the command-line arguments are in `argv[1]` to `argv[argc-1]`

`argv[argc]` is usually a NULL (zero) pointer

command-line arguments

this program prints

- the name used to run it, and
- the number of command-line arguments

```
#include <stdio.h>

int main(int argc, char *argv[])
{
  printf("%s + %d\n", argv[0], argc);
  return 0;
}
```

pointers to functions

like arrays, functions are not variables (you cannot assign to a function name)

like arrays, the value of the name of a function is a pointer to the function

you can declare a pointer to a function, which *is* a variable

a pointer to a function can be called like any other function

```
int add(int x, int y) { return x + y; }
int sub(int x, int y) { return x - y; }

int (*operator)(int, int);  // pointer to function

  operator = adding ? add : sub;
  int result = operator(3, 4);
```

complicated declarations

declarations look exactly like the use of the variable they declare

three operators can appear in declarations (and their subsequent uses):

- the pointer indirection operator: $*x$
- function call: $x()$
- array indexing: $x[]$

$*$ associates right-to-left and has lower precedence than the other two

$()$ and $[]$ associate left-to-right and have the same precedence

any part of a declaration can be parenthesised to change the order of precedence

to read a declaration, start at the name and work outwards to the base type on the left

```
int *f();     // function returning pointer to int
int (*pf)(); // pointer to function returning int
```

complicated declaration examples

`char **argv;`	`argv` is pointer to pointer to `char`
`int (*table)[13];`	`table` is pointer to array[13] of `int`
`int *table[13];`	`table` is array[13] of pointer to `int`
`void *comp();`	`comp` is function returning pointer to `void`
`void (*comp)();`	`comp` is pointer to function returning `void`
`char (*(*x())[])();`	`x` is function returning pointer to array[] of pointer to function returning `char`
`char (*(*x[3])())[5];`	`x` is array[3] of pointer to function returning pointer to array[5] of `char`

each function parameter is a declaration, as described above, e.g:

`int (*f)(int (*)(int))`	`f` is a pointer to a function returning `int` with one parameter (a pointer to function with `int` parameter returning `int`)

next week: structures, recursive structures, bit fields

`point` structure
`rect` structure
`makepoint()`, `addpoint()`, `ptinrect()`, `canonrect()`
pointers to structures and `(*x).y` vs. `x->y`
`binsearch()` for strings; counting keywords in text
`sizeof()` and counting number of array elements
pointer version of `binsearch()`
binary tree implementation
hash table implementation
using `typedef` for tree nodes
unions and bit fields

Chapter 6. Structures
6.1 Basics of Structures
6.2 Structures and Functions
6.3 Arrays of Structures
6.4 Pointers to Structures
6.5 Self-referential Structures
6.6 Table Lookup
6.7 Typedef
6.8 Unions
6.9 Bit-fields

assignment

please download assignment from

MS Team “Information Processing 2”, “General” channel

exercise preparation: dynamic storage allocation

so far all our arrays have been allocated statically

- their sizes are known at compile time, and fixed
- the compiler reserves space for them

when processing input (especially text) sizes are now known at compile time

- storage for arrays or strings must be obtained at run time

`p = malloc(size)` dynamically allocates `size` bytes of storage

- return value is the address of (a pointer to) the storage

`free(p)` de-allocates the storage pointed to by `p`

- the storage must have been allocated by `malloc()`
- you must not try to use the storage after it has been `free`d

exercise preparation: dynamic storage allocation

```c
#include <stdlib.h>   // malloc(), free()
```

```c
int strlength(char *s) {
  char *p = s;
  while (*p) ++p;
  return p - s;
}
```

```c
char *strcopy(char *s, char *t) {
  char *r = s;
  while ((*s++ = *t++)) ;
  return r;
}
```

```c
char *strduplicate(char *s)
{
  int len = strlength(s);
  char *copy = malloc(len + 1);
  strcopy(copy, s);
  return copy;
}
```

exercise preparation: dynamic storage allocation

the standard C library has its own versions of these functions

- practice using them now

they are declared in `string.h` so include it at the start of your program:

```
#include <string.h>
```

three functions are particularly useful:

`strlen(char *s)` returns the length of `s` (excluding nul terminator)

`strcpy(char *s, char *t)` copies string `t` to string `s`

`strdup(char *s)` copies `s` into newly allocated storage

- approximately: `strcpy(malloc(strlen(s) + 1), s)`

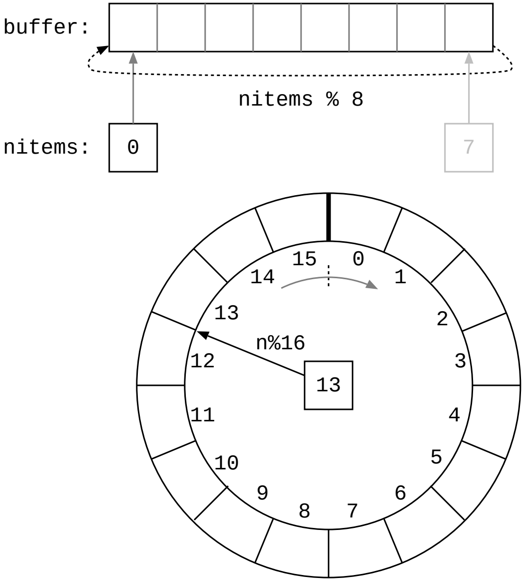
exercise preparation: dynamic storage allocation

```
#define LINEMAX  1024
#define LINESMAX 1024

int main()
{
  char line [LINEMAX], *lines[LINESMAX];
  int nlines = 0;

  while (nlines < LINESMAX && getchars(line, sizeof(line)) >= 0) {
    lines[nlines++] = strdup(line);
  }

  while (nlines-- > 0) {
    printf("%s\n", lines[nlines]);
    free(lines[nlines]);
  }

  return 0;
}
```

exercise preparation: ‘tail’ program

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

int getchars(char *s, int limit)
{
    int i = 0, c;
    while (i < limit - 1 && EOF != (c = getchar()) && c != '\n')
        s[i++] = c;
    if (EOF == c && i == 0) return -1;
    s[i] = '\0';
    return i;
}
```

exercise preparation: ‘tail’ program



need to store most recent N lines of text

use an array

- write to index 0, 1, 2, ..., $N-1$
- after $N-1$ start again at 0

can use line number modulo N

visualisation: ‘circular’ array

- after index $N-1$ comes index 0

exercise preparation: ‘tail’ program

```
#define LINEMAX 1024

int main(int argc, char **argv)
{
    char  line[LINEMAX];
    char *lines[10];      // N=10 ⇒ use lines[INDEX % 10]
    int   nlines = 0;

    while (getchars(line, sizeof(line)) >= 0) {   // line available
        if (nlines > 9) free(lines[nlines % 10]); // line saved earlier
        lines[nlines++ % 10] = strdup(line);      // save current line
    }

    for (int i = nlines > 9 ? nlines - 10 : 0;  i != nlines;  ++i) {
        printf("%s\n", lines[i % 10]);
        free(lines[i % 10]); // i % 10 ensures index is in range 0...9
    }

    return 0;
}
```

exercise preparation: scanning command-line arguments

```
for (int i = 1;  i < argc;  ++i) {
  if ('-' == *argv[i]) {
    printf("- arg is %s\n", argv[i] + 1);
    continue;
  }
  if ('+' == *argv[i]) {
    printf("+ arg is %s\n", argv[i] + 1);
    continue;
  }
  printf("illegal argument\n");
  exit(1);
}
```

exercises

character arrays vs. arrays of strings

the `echo` program

`echo` with pointers

command line argument to integer value

default argument values

command line options starting with ‘`-`’

using dynamically allocated storage

command line arguments in `tail`

skipping lines at the start of the input

multiple command-line arguments

evaluating expressions read from the command line

12:40	5	last week
12:45	5	reverse lines in string
12:50	5	reverse lines in pointer array
12:55	5	test program
13:00	5	pointer version
13:05	5	multi-dimensional arrays
13:10	5	initialisation of pointer arrays
13:15	5	pointers vs. multi-dimensional arrays
13:20	5	command-line arguments
13:25	5	pointers to functions
13:30	5	complicated declarations
13:35	5	declaration examples
13:40	5	next week
13:45	10	prep: dynamic storage allocation
13:55	10	prep: tail program
14:05	5	prep: scanning command-line arguments
14:10	10	*break*
14:20	90	exercises
15:50	00	end