Arrays, Pointers, and Strings II

Jin-Soo Kim
(*jinsoo.kim@snu.ac.kr*)

Systems Software & Architecture Lab.

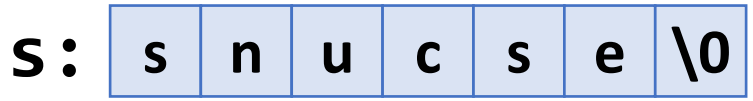
Seoul National University

Spring 2023

Strings (1)

- One-dimensional array of type `char`
- Terminated by the end-of-string sentinel `'\0'` (null character 0x00)
- The size of a string must include the storage for the null character
  - `"abcde"` : a character array of size 6

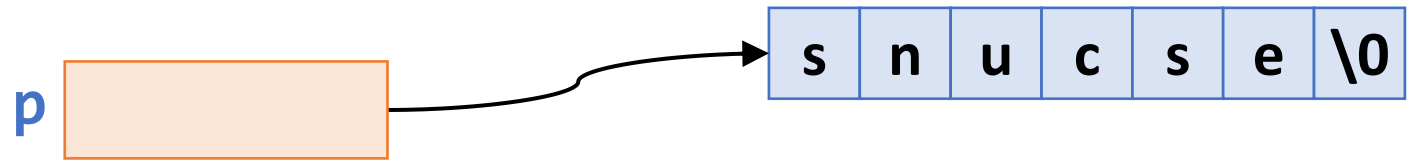
- `char s[] = "snucse";`
  ⇔ `char s[] = {'s','n','u','c','s','e','\0'};`
  - Allocates 7 bytes of memory for the array `s`



Strings (2)

- A string constant is treated with a pointer whose value is the base address of the string

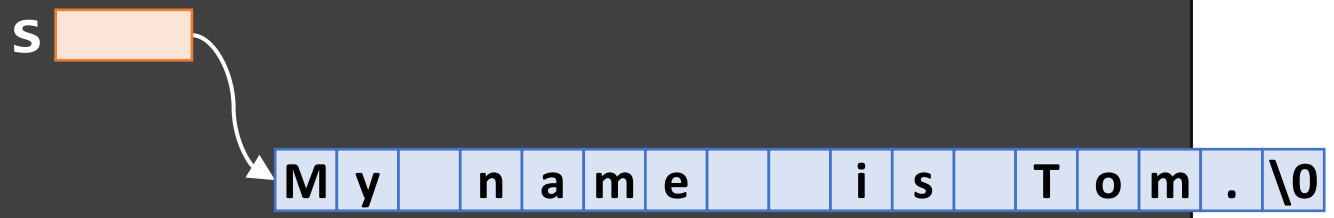
- `char *p = "snucse";`
  - Allocates space in memory for `p`
  - Puts the string constant `"snucse"` in memory somewhere else
  - Initializes `p` with the base address of the string constant



- `printf("%s %s\n", p, p+3);` /* snucse cse printed */

Example: Counting Words

```c
#include <ctype.h>

int word_count(const char *s)
{
    int cnt = 0;

    while (*s != '\0')
    {
        while (isspace(*s))      /* skip white space */
            s++;
        if (*s != '\0') {        /* found a word */
            cnt++;
            while (!isspace(*s) && *s != '\0')  /* skip the word */
                s++;
        }
    }
    return cnt;
}
```



String Functions (1)

- `#include <string.h>`
- `char *strcat(char *dest, const char *src);`
  - A copy of *src* is appended to the end of *dest*
- `char *strcpy(char *dest, const char *src);`
  - Copies the string *src* to the string *dest* including `'\0'`
- `char *strncpy(char *dest, const char *src, size_t n);`
  - Similar to `strcpy()`, but at most *n* bytes of *src* are copied

- It is programmer's responsibility to allocate sufficient space for the strings that are passed as *dest* arguments to these functions!

String Functions (2)

- `size_t strlen(const char * s);`
  - The number of characters before `'\0'`
  - The type `size_t` is an integral unsigned type
- `char *strcmp(const char * s1, const char * s2);`
  - Compares the two strings *s1* and *s2*
  - Returns: negative integer if *s1* is less than *s2*
    zero if *s1* matches *s2*
    positive integer if *s1* is greater than *s2*
- `char *strncmp(const char * s1, const char * s2, size_t n);`
  - Similar to `strcmp()`, but compares only the first (at most) *n* bytes of *s1* and *s2*

String Functions (3)

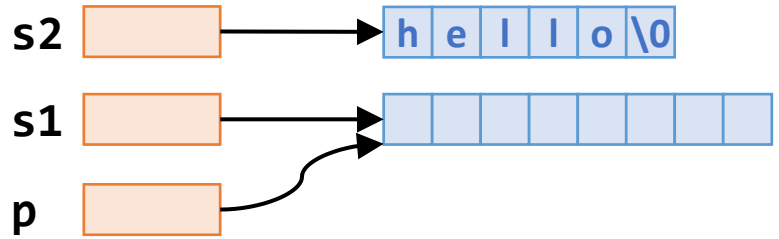
```c
size_t strlen(const char *s) {
    size_t n;
    for (n = 0; *s != '\0'; s++, n++);
    return n;
}
```

```c
char *strcpy(char *s1, const char *s2) {
    char *p = s1;
    while (*p++ = *s2++);
    return s1;
}
```

```c
char *strcat(char *s1, const char *s2) {
    char *p = s1;
    while (*p) p++;          /* go to the end */
    while (*p++ = *s2++);    /* copy */
    return s1;
}
```

`*p++` ⇔ `*(p++)`
: `p` itself is being incremented

`(*p)++`
: would increment what `p` is pointing to



`while (*p)`
⇔ `while (*p != '\0')`

String Functions (4)

Declarations and initializations	
`char s1[] = "beautiful big sky country";` `char s2[] = "how now brown cow";`	
Expression	**Value**
`strlen(s1)`	25
`strlen(s2 + 8)`	9
`strcmp(s1, s2)`	negative integer
Expression	**What gets printed**
`printf("%s", s1 + 10);`	big sky country
`strcpy(s1 + 10, s2 + 8);`	
`strcat(s1, "s!");`	
`printf("%s", s1);`	beautiful brown cows!

Multidimensional Arrays (1)

- C language allows multidimensional arrays, including arrays of arrays
- Two-dimensional array: using two bracket pairs `[][]`
  - `int a[100];` /* a one-dimensional array */
  - `int b[2][7];` /* a two-dimensional array */
  - `double c[5][3][2];` /* a three-dimensional array */
- *k*-dimensional array
  - Allocates space for S_1 x S_2 x ... x S_k elements, where S_i represents the size of the *i*-th dimension
  - Starting at the base address of the array, all elements are stored contiguously in memory

Multidimensional Arrays (2)

- Two-dimensional array: `int a[3][5];`
  - `a[i]`: the address of the `i`-th row
  - The base address of the array is `&a[0][0]`
  - Starting at the base address of the array, compiler allocates for 15 integers
  - Expressions equivalent to `a[i][j]`

```
*(a[i] + j)
(*(a + i))[j]
*((*(a + i)) + j)
*(&a[0][0] + 5*i + j)
```

a:

a[0][0]	a[0][1]	a[0][2]	a[0][3]	a[0][4]
a[1][0]	a[1][1]	a[1][2]	a[1][3]	a[1][4]
a[2][0]	a[2][1]	a[2][2]	a[2][3]	a[2][4]

Multidimensional Arrays (3)

- Storage mapping function: mapping between pointer values and array indices
  - `int a[3][5]`
    ➔ `a[i][j]: *(&a[0][0] + 5 * i + j)`
  - `int a[7][9][2]`
    ➔ `a[i][j][k]: *(&a[0][0][0] + 9 * 2 * i + 2 * j + k)`

- All sizes except the FIRST must be specified so that the compiler can determine the correct storage mapping function

Multidimensional Arrays (4)

- Initialization
  - The indexing is by rows
  - All sizes except the first must be given explicitly

  - `int a[2][3] = {1, 2, 3, 4, 5, 6};`
  - `int a[2][3] = {{1, 2, 3}, {4, 5, 6}};`
  - `int a[][3] = {{1, 2, 3}, {4, 5, 6}};`
  - `int a[][3] = {{1}, {4, 5}};`
    ⇔ `int a[][3] = {{1, 0, 0}, {4, 5, 0}};`
  - `int a[2][3] = {0};`

Multidimensional Arrays (5)

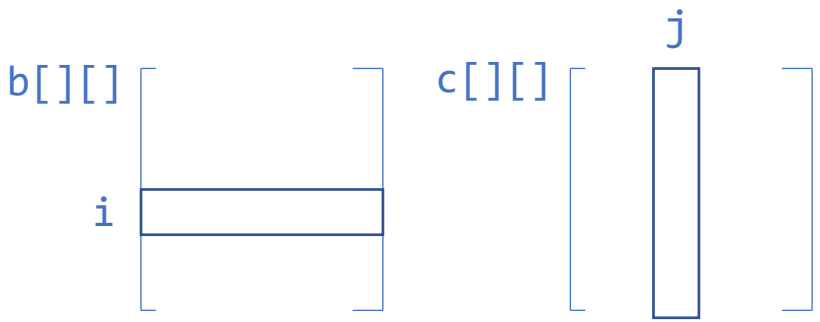
- Formal parameter declarations
  - When a multidimensional array is a formal parameter in a function definition, all sizes except the first must be specified as well

```c
int sum(int a[][5])
{
    int i, j, sum = 0;

    for (i = 0; i < 3; i++)
        for (j = 0; j < 5; j++)
            sum += a[i][j];
    return sum;
}
```

```c
int main(void)
{
    int a[3][5];

    ...

    printf("%d\n", sum(a));
    ...
}
```

Use of typedef

```
#define     N       3
typedef double  scalar;
typedef scalar  vector[N];
typedef scalar  matrix[N][N];
```



$$a_{ij} = \sum_{k=0}^{N} b_{ik} \times c_{kj}$$

```
void add(vector x, vector y, vector z) {
    int i;
    for (i = 0; i < N; i++)
        x[i] = y[i] + z[i];
}
scalar dot_product(vector x, vector y) {
    int i;
    scalar sum = 0.0;
    for (i = 0; i < N; i++)
        sum += x[i] * y[i];
    return sum;
}
void multiply(matrix a, matrix b, matrix c) {
    int i, j, k;
    for (i = 0; i < N; i++)
        for (j = 0; j < N; j++) {
            a[i][k] = 0.0;
            for (k = 0; k < N; k++)
                a[i][j] += b[i][k] * c[k][j];
        }
}
```

Arrays of Pointers

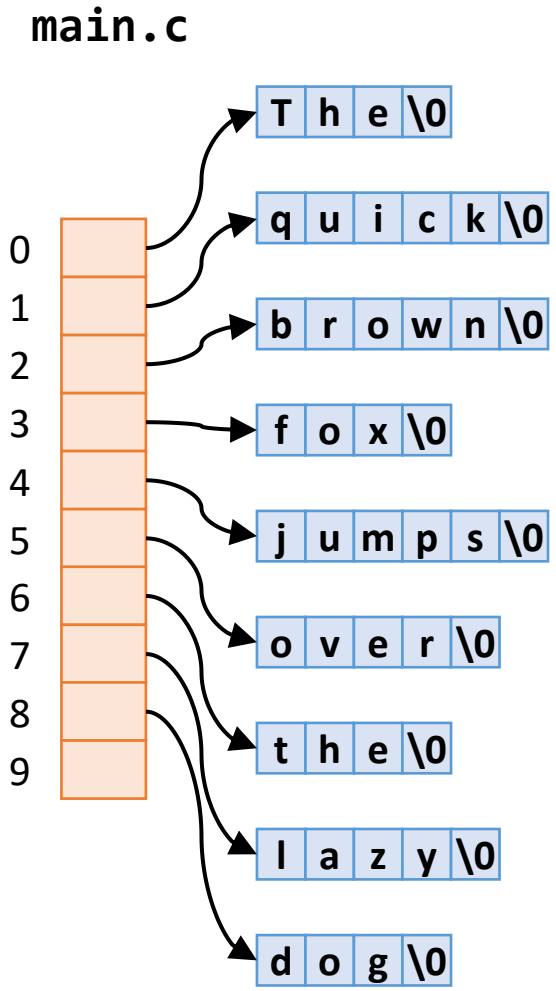
- Array elements can be of any type, including a pointer type
- Example: lexicographically sorting words in a file
  - Input: The quick brown fox jumps over the lazy dog

Output:
The
brown
dog
fox
jumps
lazy
over
quick
the

- A string array: `char *w[];`
  - Array with elements of which type is a character pointer, `char *`

Example: Sorting Words (1)

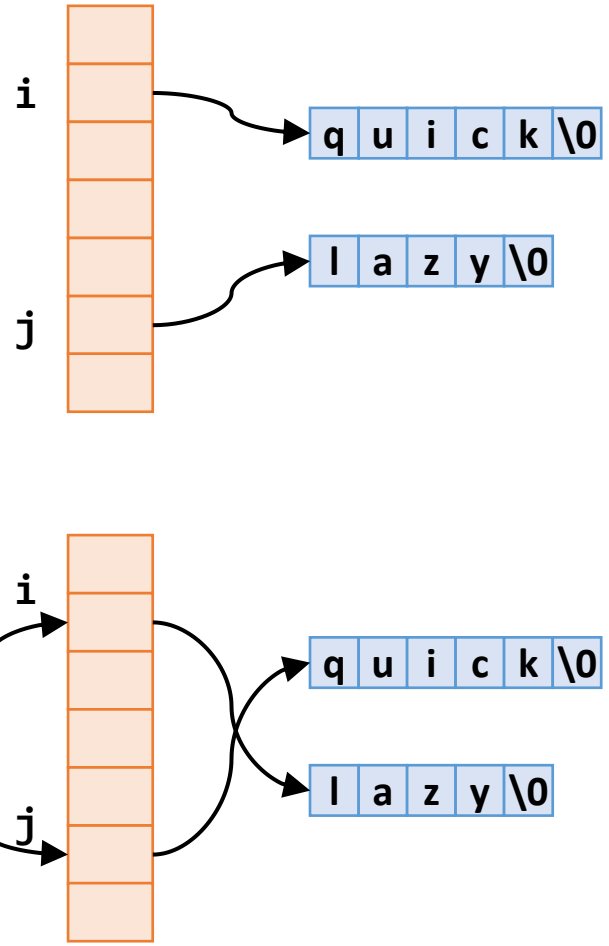
```c
#include "sort.h"

int main(void) {
    char word[MAXWORD];      /* work space */
    char *w[N];              /* an array of pointers to words */
    int n;                   /* number of words to be sorted */
    int i;

    for (i = 0; scanf("%s", word) == 1; i++)
    {
        w[i] = calloc(strlen(word) + 1, sizeof(char));
        strcpy(w[i], word);
    }
    n = i;
    sort_words(w, n);        /* sort the words */
    print_words(w, n);       /* print sorted list of words */
    return 0;
}
```



Example: Sorting Words (2)

```c
#include "sort.h"

void sort_words(char *w[], int n) {
    int i, j;

    for (i = 0; i < n; i++)
        for (j = i + 1; j < n; j++)
            if (strcmp(w[i], w[j]) > 0)
                swap(&w[i], &w[j]);
}

void swap(char **p, char **q) {
    char *tmp;

    tmp = *p;
    *p = *q;
    *q = tmp;
}
```

sort.c



Example: Sorting Words (3)

sort.h

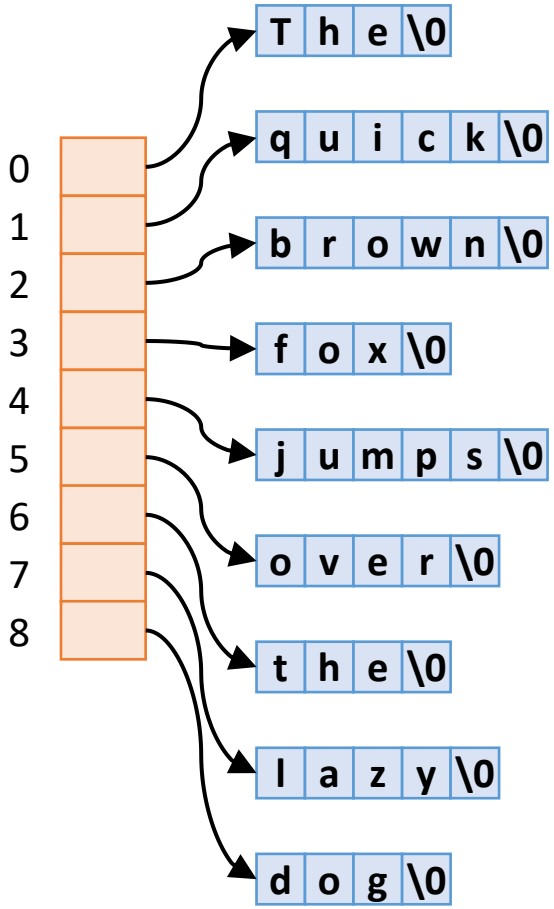
```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define MAXWORD     50      /* max word size */
#define N           300     /* array size of w[] */

void sort_words(char *w[], int n);
void swap(char **p, char **q);
void print_words(char *w[], int n);
```

print.c

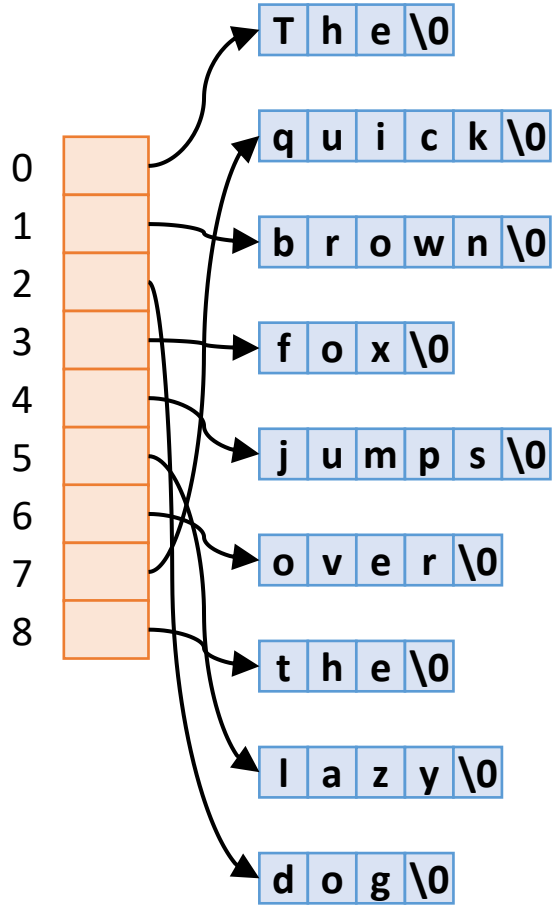
```c
#include "sort.h"

void print_words(char *w[], int n) {
    int i;
    for (i = 0; i < n; i++)
        printf("%s\n", w[i]);
}
```

Example: Sorting Words (4)

- Before sorting



- After sorting



Arguments to main()

- Two arguments, `argc` and `argv`, can be used with `main()`

```c
#include <stdio.h>

/* Echoing the command line arguments */
int main(int argc, char *argv[])
{
    int i;

    printf("argc = %d\n", argc);
    for (i = 0; i < argc; i++)
        printf("argv[%d]: %s\n", i, argv[i]);

    return 0;
}
```

```
$ gcc -o arg arg.c
$ ./arg a is for apple
argc = 5
argv[0]: ./arg
argv[1]: a
argv[2]: is
argv[3]: for
argv[4]: apple
$
```

Ragged Arrays (1)

- Ragged array
  - An array of pointers whose elements are used to point to arrays of varying sizes

```c
#include <stdio.h>

int main(void) {
    char a[2][15] = {"abc:", "a is for apple"};
    char *p[2] = {"abc:", "a is for apple"};

    printf("%c%c%c %s %s\n", a[0][0], a[0][1], a[0][2], a[0], a[1]);
    printf("%c%c%c %s %s\n", p[0][0], p[0][1], p[0][2], p[0], p[1]);
    return 0;
}
```

  - Output:
```
abc abc: a is for apple
abc abc: a is for apple
```

Ragged Arrays (2)

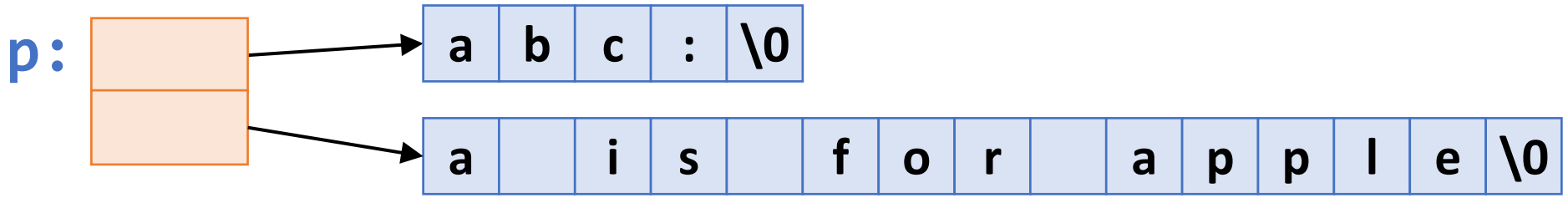
- `char a[2][15] = {"abc:", "a is for apple"};`
  - ⇔ `char a[2][15] = {{'a','b','c',':','\0'},`
    `{'a',' ','i','s', ..., '\0'}};`
  - Space for 30 chars is allocated
  - Each of `a[0]` and `a[1]` is an array of 15 chars
  - `a[0]` and `a[1]` are strings
  - Compiler generates a storage mapping function for accessing array element `a[i][j]`

`a:`

a	b	c	:	\0										
a		i	s		f	o	r		a	p	p	l	e	\0

Ragged Arrays (3)

- `char *p[2] = {"abc:", "a is for apple"};`
  - One-dimensional array of pointers to char
  - It causes space for two pointers to be allocated
  - `p[0]` is initialized to point `"abc:"`, a string constant of 5 chars, thus there is no way to modify `"abc:"` (e.g. `p[0][3] = 'd'` is not allowed)
  - `p[1]` is initialized to point at `"a is for apple"`, a string constant of 15 chars
  - `p` does its work in less space than `a`
  - Compiler does not generate a storage mapping function for accessing array elements ➔ faster working than `a`



Function as Arguments (1)

- We calculate $\sum_{x=m}^{n} f^2(x)$ for a variety of functions:
  - $f(x) = x^2 - 7x + 5$
  - $f(x) = \sin(x)$
  - $f(x) = e^{-3x/7} - 5x^2$

```c
#include <stdio.h>
#include <math.h>

double sum_square(double f(double), int m, int n) {
    int k;
    double y, sum = 0.0;

    for (k = m; k <= n; k++)
        y = f((double) k), sum += y * y;
    return sum;
}
```

Function as Arguments (2)

```c
double polynomial(double x)
{
    return (x*x - 7*x + 5);
}

double f3(double x)
{
    return (exp(-3*x/7) - 5*x*x);
}

int main(void) {
    printf("f(x) = x*x-7*x+5: %e\n", sum_square(polynomial, 1, 100));
    printf("f(x) = sin(x): %e\n", sum_square(sin, 2, 13));
    printf("f(x) = exp(-3*x/7)-5*x*x: %e\n", sum_square(f3, 0, 10));
    return 0;
}
```

Function Pointers

- `f` — a pointer to a function
  `*f` — the function itself
  `f(k)` or `(*f)(k)` — the call to the function

```c
double sum_square(double f(double), int m, int n)
/* == double sum_square(double (*f)(double), int m, int n) */
{
    int k;
    double y, sum = 0.0;

    for (k = m; k <= n; k++) {
        y = f((double) k);      /* y = (*f)((double) k); */
        sum += y * y;
    }
    return sum;
}
```

Function Parameter in Prototypes

- A number of equivalent ways to write functions as formal parameters in function prototypes

```
double sum_square(double f(double x), int m, int n);
double sum_square(double f(double), int m, int n);
double sum_square(double f(double), int, int);
double sum_square(double (*f)(double), int, int);
double sum_square(double (*)(double), int, int);
double sum_square(double g(double), int, int);
```

const

- A type qualifier that restricts or qualifies the way an identifier of a given type can be used

Usage	Meaning
`const int i;`	`i` is a constant integer
`const int *p;` `int const *p;`	`p` is a pointer to a constant integer (`p` is mutable, but `*p` is immutable)
`int * const p;`	`p` is a constant pointer to an integer (`p` is immutable, but `*p` is mutable)
`const int * const p;` `int const * const p;`	`p` is a constant pointer to a constant integer (both `p` and `*p` are immutable)

const vs. #define

```c
const double KM_PER_MILE = 1.609;

double mile2km(double mile) {
    return mile * KM_PER_MILE;
}
```

```c
#define KM_PER_MILE     1.609

double mile2km(double mile) {
    return mile * KM_PER_MILE;
}
```

- `#define` is a preprocessor directive, but const is a keyword
- `#define` is not scope-controlled
- The same identifier can be redefined using `#undef` and `#define`
- `const int` cannot be used to specify the array dimension
  - ```c
    const int n = 3;
    int a[n] = {1, 2, 3};     /* wrong */
    ```
    ```

Summary: Specifying Types

- Basic data types: char / short / int / long / long long / float / double