Computer Programming

Arrays

Marius Minea

marius@cs.upt.ro

28 October 2013

Declaring arrays

array = sequence of elements of the *same type*

In math, a *sequence* x_n is a function $x(n)$ over the naturals $\mathbb{N}$
⇒ an array `x` associates a value `x[n]` to an index `n`

Declaration: *type arrayname*[*elementcount*] ;

```
double x[20];   int mat[10][20];
```

Initialization: comma-separated elements between braces:

```
int a[4] = { 0, 1, 4, 9 };
```

Remember: local variables are *not initialized by default!*

array *size* (element count) = a positive *constant*
since C99: also variable dimension if known at declaration time

```
void f(int n) { int tab[n]; /* n known at call */}
```

Syntax *type* a[*dim*] ; suggests that a[*index*] has given *type*

Using arrays

An array *element* *arrname*[*index*] can be used as any *variable*
has a value, may be used in expressions
is an *lvalue*, may be assigned

```
x[3] = 1; n = a[i]; t[i] = t[i + 1]
```

index may be any *expression* with *integer* value

IMPORTANT! In C, array indices start at 0, end at length - 1

`int a[4];` has `a[0]`, `a[1]`, `a[2]`, `a[3]`, there is no `a[4]`

Sample array traversal and assignment:

```
int a[10]; for (int i = 10; i--;) a[i] = i + 1;
```

or forward:

```
for (int i = 0; i < 10; ++i) a[i] = i + 1;
```

NOT: `a[i] = i++;` (why?)

Named constants as array sizes

Useful to define *macro* names for constants like array dimensions

`#define NAME` *constval*

the *C preprocesssor* replaces `NAME` in the source with *constval* before compilation

Macro names: usually in `ALL CAPS` (to distinguish from vars)

```
#define LEN  30
double t[LEN];
// tabulate sin with step 0.1
for (int i = 0; i < LEN; ++i)
  printf("%f ", t[i] = sin(0.1*LEN));
```

Easier to read, occurrence of `LEN` suggests it's the array size

Program is *easier to maintain*: size only needs changed in *one place*
⇒ avoid forgetting to change it somewhere

Computing the first primes

```
#include <stdio.h>
#define MAX   100   // preprocessor replaces MAX with 100

int main(void) {
  unsigned p[MAX] = {2};      // 2 is first prime
  unsigned cnt = 1, n = 3;    // one prime; 3 is candidate
  do {
    unsigned maxdiv = sqrt(n);   // stop trying here
    for (int j = 0; n % p[j]; ++j) // while not divisible
      if (p[j] >= maxdiv) {   // can't have larger divisor
        p[cnt++] = n; break;  // store prime, exit cycle
      }
    n += 2;                   // try next odd number
  } while (cnt < MAX);        // until table full
  for (int j = 0; j < MAX; ++j)
    printf("%d ", p[j]);
  putchar('\n');
  return 0;
}
```

Variables and addresses

Any variable `x` has an *address* where its value is stored in memory

Address of `x` is obtained with the prefix *operator &* `&x`
Operand of `&`: any *lvalue* (assignable object):
variable, array element, structure field

expressions (generally), constants are not lvalues, have no address

The name of an array is the address of the array.

`int a[6];` name `a` is the array *address*
The name `a` does *NOT represent all array elements!*

Addresses may be printed (in hex) with %p format in `printf`

```
#include <stdio.h>
int main(void) {
  double d; int a[6];
  printf("Address of d: %p\n", &d); // & for address
  printf("Address of a: %p\n", a); // a is an address
  return 0;
}
```

Arrays in C and other languages

In C, an array name represents *just its address*
not the block of its elements!

exception: `sizeof(`*arrname*`)` is *elemcnt* `*` `sizeof(`*elemtype*`)`

The address carries no information about the array size!

In other languages, an array is an *object*
carries the *length information with it*
`Array.length` *property* in C# (*field* in Java)
⇒ having an array, one can immediately find out its length
⇒ can implement bounds checks, etc.

C has none of that!
Keeping track of the array size is the *programmer's responsibility*
⇒ lots of room for *error!*

Arrays as function parameters

As function argument, the *address* of the array is passed

carries *NO length information*

⇒ typically, length is given as another parameter

DON'T write `[length]` in parameter declaration, does not matter

only confuses reader;

neither compiler nor runtime can check or know length!

```
#include <stdio.h>
void printtab(int t[], unsigned len)
{
  for (int i = 0; i < len; ++i) printf("%d ", t[i]);
  putchar('\n');
}
int main(void)
{
  int prime[10] = {2, 3, 5, 7, 11, 13, 17, 19, 23, 29 };
  printtab(prime, 10); // NOT prime[10], NOT prime[]
  return 0;
}
```

Arrays as function parameters

In C, arguments are passed by value

⇒ also applies to arrays: *value of address* is passed
But: having address, the function may *read* and *write* array elements

```
void sumvec(double a[], double b[], double r[], size_t len)
{
  for (unsigned i = 0; i < len; ++i) r[i] = a[i] + b[i];
}
#define LEN 3 // macro for array length
int main(void) {
  double a[LEN] = {0, .5, 1}, b[LEN] = {1, .7, 1}, c[LEN];
  sumvec(a, b, c, LEN);
  return 0;
}
```

Initialization

Uninitialized arrays have undefined values (error if used!)
Partially initialized arrays have remaining elements set to zero

Computing an aggregate value from an array

```
double sum(double a[], unsigned len)
{
  double s = 0.;                  // must be initialized
  for (unsigned i = len; i--;) // in any direction
    s += a[i];
  return s;
}

#define LEN 4

int main(void)
{
  double a[LEN] = { 1.0, 2.3, -5.6, 7 };
  printf("%f\n", sumtab(a, LEN));
  return 0;
}
```

Accumulated result (s) *must be initialized*

Direction of traversal may matter or not, depending on the problem

Selective processing of array elements

```
// average of passing grades
double pass_avg(double a[], unsigned len)
{
  double s = 0.;       // initialize sum
  unsigned num = 0;    // count selected elements
  for (unsigned i = len; i--;)
    if (a[i] >= 5) {  // only for passing grades
      s += a[i];
      ++num;
    }
  return num ? s / num : 0; // return 0 if none passed
}
```

Division by 0 would return `NaN` (not a number, `math.h`)

$\Rightarrow$ we return a value (0) distinct from any normal result (≥ 5)

Searching for an array element

Which is the smallest prime factor of n ≤ 1000 ?
(we have all primes p with $p^2 \leq 1000$)

```
#define NP 11
unsigned ptab[NP] = {2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31};

unsigned factor(unsigned n)
{
  for (unsigned i = 0; i < NP; ++i)
    if (n % ptab[i] == 0) return ptab[i]; // prime factor
  return n;    // no prime factor <= 31, n is prime
}
```

In this *pattern* we search the *first* element satisfying a condition.
Once found, no need to search further: exit function with `return`
If loop exits normally (nothing found), return some other value (n)

If using `break`, need to check afterwards reason for loop exit (normal or forced), perhaps setting a flag; easier with `return`

Searching for an array element

Use **break** when we only want to exit loop, not function
Before loop, initialize result with value signaling search failure
(can then check whether search was successful)

```
#include <stdio.h>

#define NP 11
unsigned ptab[NP] = {2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31};

int main(void)
{
  unsigned n = 751, p = n;   // p = n means n prime
  for (unsigned i = 0; i < NP; ++i)
    if (n % ptab[i] == 0) { p = ptab[i]; break; }
  if (p < n) printf("%u is least prime factor of %u\n", p, n);
  else printf("%u is prime\n", n);
  return 0;
}
```

Counting character frequencies

Count how many times each character appears in input

- `getchar()` returns unsigned char as int, fine for indexing
- DON'T USE `char` as index type (may be signed or unsigned!)
  - if needed, index with an `unsigned char`

```
#include <stdio.h>
int main(void)
{
  int c;
  unsigned frq[256] = {0};   // indexed by character
  while ((c = getchar()) != EOF)
    ++frq[c];  // frequency of c (character code)
  for (unsigned car = 0; car < 256; ++car)
    if (frq[car] != 0)
      printf("%c appears %u times\n", car, frq[car]);
  return 0;
}
```

Improve: print escape sequence `\t` `\n` etc. for non-printing chars.

Use `switch` with `default`

Variable length arrays (C99)

size must be known at declaration time (e.g., function parameter)

```c
#include <stdio.h>
#include <string.h>
void fraction(unsigned m, unsigned n) {
  int seen[n];        // size given by parameter n
  memset(seen, 0, sizeof(seen)); // initialize
  printf("%u.", m/n); // integer part
  for (; m %= n; m *= 10) {  // nonzero remainder
    putchar(10*m/n + '0');   // quotient = next digit
    if (seen[m]) { printf("..."); break; } // periodic
    seen[m] = 1;       // mark remainder as seen
  }
  putchar('\n');
}
int main(void) {
  fraction(5, 28);     // 5/28 = 0.178571428...
  return 0;
}
```

Arrays and character strings

```
char word[20];  // uninitialized char array
char name[3] = { 'C', 'T', 'I' }; // exactly 3 chars
```

In C, *strings* are character sequences terminated in memory by the `'\0'` character (code 0).

String constants `"hello\n"` also end with `'\0'`
  terminator `'\0'` (null character) takes up memory
  but is not counted as part of string length

```
char msg[] = "test";  // 5 bytes, terminated with '\0'
char msg[] = {'t','e','s','t','\0'}; // same thing
char str[20] = "test"; // remainder to 20 chars are '\0'
```

For initialized strings without explicit dimension (`msg` above), allocated size is that of initializer, plus `'\0'`

All *standard functions* for strings need *null-terminated* strings.

The pointer type

The result of an *address* operation has a *type*, like any expression

For a variable declared *type* `x;` type of address `&x` is *type* `*`
read: *type pointer*; i.e., an address of an object of that *type*

The name of an array has the type pointer to elementtype

`int a[4];` `a` has type `int *`
`char s[8];` `s` has type `char *`

In function declarations, `void f(`*type* `a[])` means `void f(`*type* `*a)`
this is why the *size is ignored* `void f(`*tip* `a[6])`

The value `NULL` (0 of type `void *`, address of unspecified type) indicates an *invalid address* (used when one needs an address value, but there is no valid address)

A string is (has type) `char *`

In C, a *string* is represented by *its address*, it is a `char*`

including string *constants*: `"something"`

CAUTION! `'a'` is a `char`, but `"a"` is a string (`char *`)

`char` and `char *` are completely different things!

A string (constant or not) is null-terminated (`'\0'`)

Functions that work with strings can thus know where strings end (no need for an extra length parameter)
BUT: to compute string length must look at all chars (expensive)

CAUTION! Compare strings with `strcmp`, `strncmp`, NOT with ==
== compares *addresses* (WHERE strings are), NOT their contents
BUT: could use singleton strings for efficient comparison

CAUTION! a string *constant* `"test"` CANNOT be modified
(do not pass it to a function that modifies its argument)

String functions (string.h)

```
size_t strlen(const char *s); // length until \0
char *strchr(const char *s, int c); // find char c in s
char *strstr(const char *big, const char *small); // find str
// both return address where found, or NULL if not found

int strcmp (const char *s1, const char *s2);
// returns int < 0 or 0 or > 0 (order of s1 and s2)
int strncmp (const char *s1, const char *s2, size_t n);
// compares over length at most n

char *strcpy(char *dest, const char *src); // copy src to dest
char *strcat(char *dest, const char *src); // dest concat src
// DANGER, OVERFLOW if not enough space at dest
// strcat inefficient for repeat append of short string to long
char *strncpy(char *dest, const char *src, size_t n);
char *strncat(char *dest, const char *src, size_t n);
// copies/appends at most n chars from src to dest
```

`size_t`: unsigned integer type for sizes of objects

`const`: type qualifier: object will not be changed

String functions (cont'd)

```
void *memset(void *s, int c, size_t n);
// fills memory with n bytes of byte value c
void *memcpy(void *dest, const void *src, size_t n);
// copies n bytes from src to dest; areas can't overlap
void *memmove(void *dest, const void *src, size_t n);
// moves n bytes from src to dest; areas may overlap

size_t strspn(const char *s, const char *accept);
// counts initial length of s made up from chars in accept
size_t strcspn(const char *s, const char *reject);
// counts initial length of s made up from chars not in reject
```

Multidimensional arrays (matrices)

Arrays with elements that are themselves arrays (matrix lines)

Declaration: *type name*[*dim1*][*dim2*]...[*dimN*];

Example: `double m[6][8]; int a[2][4][3];`
`m`: array of 6 elements, each an array of 8 reals
Addressing an element: `m[4][3]`

Dimensions: *constant* (since C99: known at declaration point)

Array elements are consecutive in memory
`m[i][j]` is in position `i*COL+j`

Traversing a matrix

```
#define LIN 2 // number of lines
#define COL 5 // number of columns
int main(void) {
  double a[LIN][COL] = { {0, 1, 2, 3, 4}, 5, 6, 7, 8, 9 };
  // inner brace groups elements of a line;
  // can also write without grouping
  for (int i = 0; i < LIN; ++i) { // iterate over lines
    for (int j = 0; j < COL; ++j) // iterate over columns
      printf("%f ", a[i][j]);
    putchar('\n');                // end each line
  }
  return 0;
}
```

Multidimensional arrays as function parameters

`m[i][j]` is in position `i*COL+j` ⇒ must know `COL`
⇒ must know *all* dimensions except first: $A_{lin \times 10} \times B_{10 \times 6} = C_{lin \times 6}$

```
void matmul(double a[][10],double b[][6],double c[][6],int lin)
  for (int i = 0; i < lin; ++i) // works only on matrices
    for (int j = 0; j < 6; ++j) { // of size 10 and 6
      c[i][j] = 0;
      for (int k = 0; k < 10; ++k) c[i][j] += a[i][k]*b[k][j];
    }
} // to use it, e.g. in main:
double m1[8][10], m2[10][6], m3[8][6]; // then assign values
matmul(m1, m2, m3, 8); // NOT m1[][], NOT m2[][6], NOT m3[8][6]
```

Better:
C99 allows variable length arrays, if length known at call time
⇒ lengths as parameters, *before* arrays that use them:

```
void matmul(int l, int n, int p, double a[][n], double b[n][p],
                  double c[][p]); // n, p declared before use
```