Computer Programming

Arrays

Marius Minea

marius@cs.upt.ro

28 October 2014

Declaring arrays

array = sequence of elements of the *same type*

In math, a *sequence* x_n is a function $x(n)$ over the naturals $\mathbb{N}$
In C, an array `x` associates a value `x[n]` to an index `n` starting at 0

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 time */}
```

Array size *MUST* be given; declaring ~~`int a[];`~~ is an *ERROR*
as local variable (and has size 1 as global variable)

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 used on left hand of assignment

`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++;`~~ (side effect and index use are unordered)

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* $\Rightarrow$ avoid forgetting to change it somewhere

Computing the first primes

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

int main(void) {
  unsigned p[MAX] = {2, 3};  // 2, 3 are first primes
  unsigned cnt = 2, n = 5;   // two primes; 5 is candidate
  do {
    unsigned maxdiv = sqrt(n);   // stop trying here
    for (int j = 1; 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;
}
```

Don't write beyond the length of an array!

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

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!*

Overflowing an array is highly dangerous! (common security flaw)

Arrays as function parameters

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

carries *NO length information*

$\Rightarrow$ 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;
}
```

Recall: parameter passing in C

In C, function arguments are passed BY VALUE

argument *expressions* are evaluated
resulting *values* of given type are given to parameters

```
#include <stdio.h>
void inc(int x) {
  printf("in function: %d", ++x); // prints 6
}
int main(void) {
  int y = 5;
  inc(y); // called with VALUE 5
  printf(" in main: %d", y); // prints 5 !
  return 0;
}
```

`inc()` is NOT called with ~~variable `y`~~. It is called with *value 5*.
It doesn't know it was called as `inc(y)`. We cannot make it know.
Parameter `x` can be assigned, but its lifetime is the function block.
Values are NOT ~~passed back through parameters~~
what would you do returning from `inc(y*y+y+7)` ???

Arrays as function parameters

In C, arguments are passed by value For arrays: *value of address*

But: having address, a function may *read* and *write* array elements

```
void sumvec(double a[], double b[], double r[], size_t len) {
  // can't create r here, since lifetime is that of function
  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); // must create c in main, pass as arg
  return 0;
}
```

Can't return array declared in function. Disappears at function end!

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;
}
```

Strings are special character arrays

```
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 (null character, code 0).

String constants `"hello\n"` also end with `'\0'`

- terminator `'\0'` uses one extra memory byte
- 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.

Strings and their length: don't loop until `strlen(s)`

Strings are character arrays terminated by `'\0'`.
This is how we can find out their length.
Not so for other arrays. We must pass the length to any function.

```
void process_array(int a[], unsigned len);
```

For strings, just pass the string, and traverse until `'\0'`

```
void process_string(char s[]) { // loops while s[i] nonzero
  for (int i = 0; s[i]; ++i) { /* do work */ }
}
unsigned strlen(const char s[]) { // will not change s
  unsigned i = 0; while (s[i]) ++i; return i; // i is length
} // strlen is a standard function in string.h
```

DON'T write ~~`for (int i=0; i < strlen(s); i++) {/*code*/}`~~
NOR ~~`int n=strlen(s); for (int i=0; i < n; ++i) {/*code*/}`~~
This runs through (long?) string to find length, then does it again.

In other languages, String.length (or the like) returns a stored value.
In C, `strlen` is an expensive run through the (maybe long!) string.

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;
}
```

The pointer type

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

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

In *elemtype tab*`[LEN];` the array name `tab` has type *elemtype* `*`

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

A function declaration *restype* `f(`*eltype* `a[])` means (becomes)
restype `f(`*eltype* `*a)`

the *size is ignored* *rtyp* `f(`*eltyp* `a[`~~6~~`])`

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

A string is (has type) `char *`

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 *strncpy(char *dest, const char *src, size_t n);
// copies at most n chars; not terminated with \0 if src longer

char *strcat(char *dest, const char *src); // dest concat src
// DANGER, OVERFLOW if not enough space at dest
char *strncat(char *dest, const char *src, size_t n);
// appends at most n chars; always adds \0 (may use n+1)
// avoid str(n)cat, inefficient: runs through dest to find end
```

String functions (cont'd)

`size_t`: unsigned integer type for sizes of objects
`const`: type qualifier: object will not be changed

```
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

```c
#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, but should be consistent
  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;
}
```

Matrices as function parameters

We can rewrite the printing as a function:

```
void mat_print(unsigned m, unsigned n, double a[m][n])
   // or double a[][n]  or double (*a)[n]
{
  for (int i = 0; i < m; ++i) {
    for (int j = 0; j < n; ++j)
      printf("%.2f ", a[i][j]);
    putchar('\n');
  }
}
```

`a[m][n]` or `a[][n]` or `double (*a)[n]` is same for compiler;
it only needs size of object at address a: a vector, `double [n]`
Write `a[m][n]` helps the reader see the dimensions
(but compiler won't warn if we pass matrix with fewer/more lines)

Matrices as function parameters (cont'd)

`m[i][j]` is in position `i*COL+j` $\Rightarrow$ must know `COL`
$\Rightarrow$ 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
$\Rightarrow$ lengths as parameters, *before* arrays that use them:

```
void matmul(int m, int n, int p, double a[m][n], double b[n][p],
            double c[m][p]); // m, n, p declared before use
// here, writing first dimension is good for code understanding
```

Common errors with arrays

Overflow by not checking loop limit:

```
int n, a[100];
printf("please input number of elements:");
scanf("%d", &n); // reads n (maybe, no check)
for (int i = 0; i < n; ++i) a[i] = i; // what if n>100 ???
```

Overflow by assuming input will end in time:

```
char s[100]; int c, i = 0;
while ((c = getchar()) != '\n')
  s[i++] = c; // what if line longer than 100 ?
```

Assuming array bound in function actually means something

```
void f(int a[100]) // we see [100], compiler sees a[]
{
  for (int i = 0; i < 100; ++i) // code written for 100
  {/* work with a[i] */} // what if array of 50 was passed?
}
```