C Programming
Arrays

Adam Sampson (based on material by Greg Michaelson)
School of Mathematical and Computer Sciences
Heriot-Watt University

`sizeof`

- We've seen some of C's built-in types already: `char`, `int`, `long`, `float`, `double`...
- `sizeof(`*`type`*`)` tells you how much memory a variable of that type would occupy
  - Looks like a function call, but actually a built-in operator
- `sizeof(char)` is **always** `1`
  - C counts sizes in `char`s – usually 8-bit bytes
- `sizeof(int)` is usually `4`; `sizeof(int *)` will be `4` on 32-bit machines, and `8` on 64-bit machines

Arrays

- Finite sequence of elements of same type,
  occupying a contiguous range of memory
- To declare an array:
  `type name[length];` e.g. `char myChars[10];`
- Allocates `length * sizeof(type)` memory
  - `char b[6];` allocates 6 * 1 == 6 bytes
  - `int a[6];` allocates 6 * 4 == 24 bytes
  - `double c[6];` allocates 6 * 8 == 48 bytes
- Element indexes are numbered from `0` to `length-1`

Array size

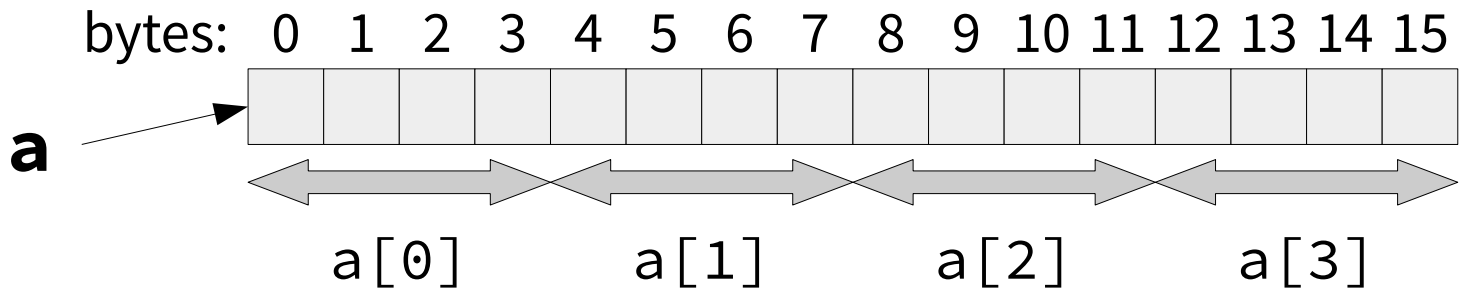
- To declare an array:

```
type name[length];
```

- ANSI C: length must be a constant integer
  - You can't decide size at run-time;
    you must declare a “too big” array
- C99: size can be computed at runtime for stack-allocated arrays – a “VLA”, variable-length array
  - Size of the stack is usually limited, so we often avoid VLAs

Array access

```
int a[4];
```



- Items:
  - `a[0]` is at `a+0*4 == a`
  - `a[1]` is at `a+1*4 == a+4`
  - `a[2]` is at `a+2*4 == a+8`
  - `a[3]` is at `a+3*4 == a+12`

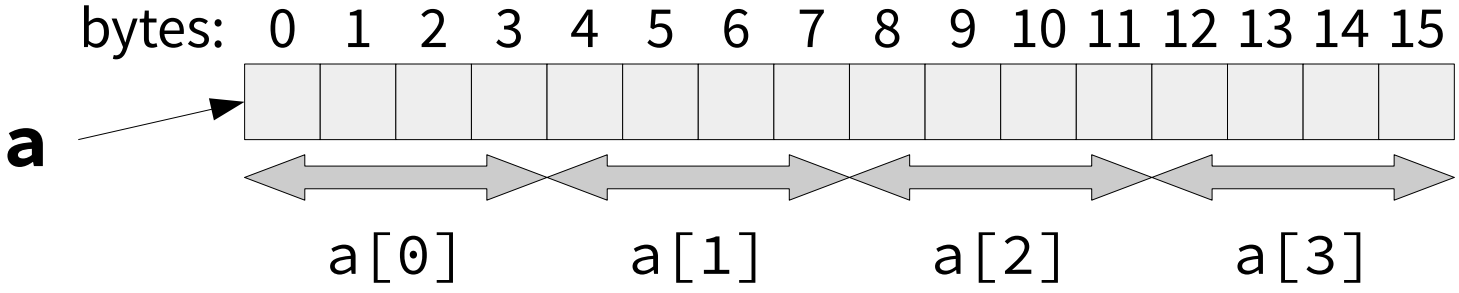
Array name and address

- name is an alias for the address of the array's first byte
  - name is not a variable (you can't change it)
- Can be used as if it's a pointer, i.e. `name == &name`

```
int a[3];
printf("a: %x; &a: %x\n", a, &a);
→ a: 80497fc; &a: 80497fc
```

a's value

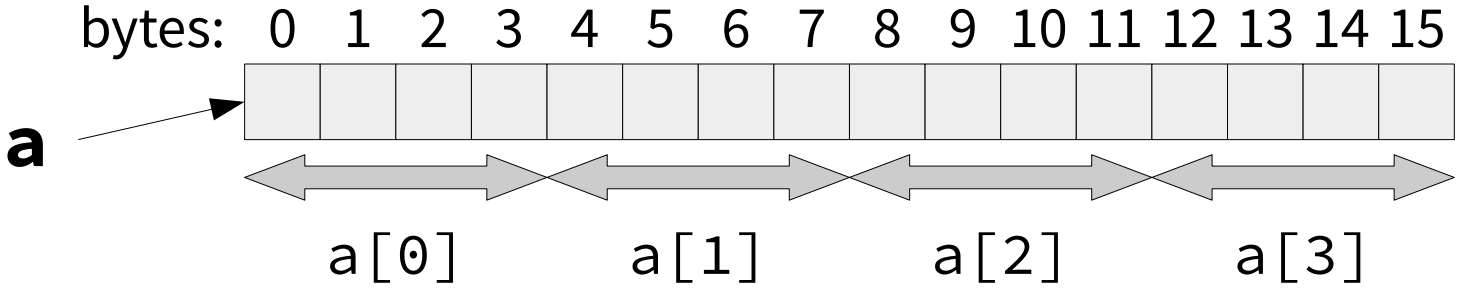
a's address



Array access: assignment

`exp1[exp2] = expression;`

- Evaluate `exp1` to get pointer
- Evaluate `exp2` to get integer index
- Evaluate `expression` to value
- Store value at location `exp1 + exp2 * sizeof(`*`type`*`)`
  - i.e. address of `exp2`-th element of *`type`* after address of first byte



Array bounds

- C has no array bounds checking
  - Array lengths are **not known** at runtime
- If you try to access array outside bounds
  - may get weird values from bytes outside array
- or
  - program may crash
- This is an example of **undefined behaviour** – compiler is allowed to assume that a correct program will never do it

Arrays as function parameters

`type name[]`

`type name[length]`

As a formal function parameter, both mean the same as:

`type * name`

By convention, `[]` is used when it must be a **non-NULL** pointer – hence `main(int argc, char *argv[])`

Preprocessor definition

```
#define name text
```

- A **macro**: the C pre-processor textually replaces all later occurrences of `name` in program with `text`, before compilation

- Used to define constants at start of program
  e.g. `#define SIZE 127`

`scalar.c`

Scalar product of two vectors

Scalar product

- Read two vectors V_0 and V_1 from a file
- Calculate $V_0[0]*V_1[0] + ... + V_0[N-1]*V_1[N-1]$

```
$ ./scalar vecs.txt
 1  2  3  4  5  6
 7  8  9 10 11 12
scalar product: 217
```

Scalar product

- File contains:
  - length of vector - N
  - 1st vector – $V_0[0]...V_0[N-1]$
  - 2nd vector – $V_1[0]...V_1[N-1]$
- Functions to:
  - read vector
  - print vector
  - calculate scalar product

Scalar product

```
#include <stdio.h>

#define MAX 100

void getVec(FILE * fin, int v[], int n)
{
    int i;
    i = 0;
    while (i < n) {
        fscanf(fin, "%d", &(v[i]));
        i = i + 1;
    }
}
```

Array parameter

Extra parameter for array length

Scalar product

```
void printVec(int v[], int n)
{
    int i;
    i = 0;
    while (i < n) {
        printf("%2d ", v[i]);
        i = i + 1;
    }
    printf("\n");
}
```

Scalar product

```
int scalar(int v0[], int v1[], int n)
{
    int s;
    int i;
    s = 0;
    i = 0;
    while (i < n) {
        s = s + v0[i] * v1[i];
        i = i + 1;
    }
    return s;
}
```

Scalar product

```
int main(int argc, char *argv[])
{
    FILE *fin;
    int v0[MAX], v1[MAX];
    int n;
    if (argc != 2) {
        printf("scalar: wrong number of arguments\n");
        return 1;
    }
    if ((fin = fopen(argv[1], "r")) == NULL) {
        printf("scalar: can't open %s\n", argv[1]);
        return 1;
    }
```

Scalar product

```
    fscanf(fin, "%d", &n);
    if (n >= MAX) {
        printf("scalar: %d vector bigger than %d\n", n, MAX);
        fclose(fin);
        return 1;
    }
    getVec(fin, v0, n);
    printVec(v0, n);
    getVec(fin, v1, n);
    printVec(v1, n);
    fclose(fin);
    printf("scalar product: %d\n", scalar(v0, v1, n));
    return 0;
}
```

Important: check we won't exceed the fixed size of the array!

C lectures

- Compiling code, program layout, printing/reading data, expressions, arithmetic, memory addresses, control flow, precedence
- **Functions, pointers, file IO, arrays**
- Memory allocation, casting, masking, shifting
- Strings, structures, dynamic space allocation, field access
- Recursive structures, 2D arrays, union types