Arrays and Pointers in C

Alan L. Cox
alc@rice.edu
Objectives

Be able to use arrays, pointers, and strings in C programs

Be able to explain the representation of these data types at the machine level, including their similarities and differences
Arrays in C

- All elements of same type – homogenous
- Unlike Java, array size in declaration

```c
int array[10];
int b;
array[0] = 3;
array[9] = 4;
array[10] = 5;
array[-1] = 6;
```

- First element (index 0)
- Last element (index size - 1)

Compare:

```c
int array[10];
```

```java
int[] array = new int[10];
```

No bounds checking!

Allowed – usually causes no *obvious* error

`array[10]` may overwrite `b`
Array Representation

Homogeneous → Each element same size – s bytes
- An array of m data values is a sequence of m×s bytes
- Indexing: 0th value at byte s×0, 1st value at byte s×1, ...

m and s are not part of representation
- Unlike in some other languages
- s known by compiler – usually irrelevant to programmer
- m often known by compiler – if not, must be saved by programmer

```c
int a[3];
```
Array Representation

Could be optimized by making these adjacent, and reducing padding (by default, not)

Array aligned by size of elements
Array Sizes

int array[10];

What is

sizeof(array[3])? 4

sizeof(array)? 40

returns the size of an object in bytes
Multi-Dimensional Arrays

Recall: no bounds checking

What happens when you write:

```c
matrix[0][3] = 42;
```

“Row Major” Organization
Variable-Length Arrays

New C99 feature: Variable-length arrays defined within functions

Global arrays must still have fixed (constant) length

```c
int function(int n) {
    int array[n];
    ...
}
```
Memory Addresses

Storage cells are typically viewed as being byte-sized

- Usually the smallest addressable unit of memory
  - Few machines can directly address bits individually
- Such addresses are sometimes called byte-addresses

Memory is often accessed as words

- Usually a word is the largest unit of memory access by a single machine instruction
  - CLEAR’s word size is 8 bytes (= sizeof(long))
- A word-address is simply the byte-address of the word’s first byte
Special case of bounded-size natural numbers
- Maximum memory limited by processor word-size
- \(2^{32}\) bytes = 4GB, \(2^{64}\) bytes = 16 exabytes

A pointer is just another kind of value
- A basic type in C

\[
\text{int \ *} \text{ptr;}
\]

The variable “ptr” stores a pointer to an “int”. 
Pointer Operations in C

Creation
& variable Returns variable’s memory address

Dereference
* pointer Returns contents stored at address

Indirect assignment
* pointer = val Stores value at address

Of course, still have...

Assignment
pointer = ptr Stores pointer in another variable
Using Pointers

int i1;
int i2;
int *ptr1;
int *ptr2;

i1 = 1;
i2 = 2;

ptr1 = &i1;
ptr2 = ptr1;

*ptr1 = 3;
i2 = *ptr2;
Using Pointers (cont.)

```
int int1 = 1036; /* some data to point to */
int int2 = 8;

int *int_ptr1 = &int1; /* get addresses of data */
int *int_ptr2 = &int2;

*int_ptr1 = int_ptr2;
*int_ptr1 = int2;
```

Type check warning: int_ptr2 is not an int

int1 becomes 8
Using Pointers (cont.)

```c
int int1 = 1036; /* some data to point to */
int int2 = 8;

int *int_ptr1 = &int1; /* get addresses of data */
int *int_ptr2 = &int2;

int_ptr1 = *int_ptr2;
int_ptr1 = int_ptr2;
```

What happens?

Type check warning: *int_ptr2 is not an int *

Changes int_ptr1 – doesn’t change int1
Pointer Arithmetic

\[ \text{pointer} + \text{number} \quad \text{pointer} - \text{number} \]

E.g., \( \text{pointer} + 1 \) adds 1 \text{something} to a pointer

\begin{align*}
\text{char} & \quad \ast p; \\
\text{char} & \quad a; \\
\text{char} & \quad b; \\
p & = \& a; \\
p & += 1; \\
\text{int} & \quad \ast p; \\
\text{int} & \quad a; \\
\text{int} & \quad b; \\
p & = \& a; \\
p & += 1;
\end{align*}

In each, \( p \) now points to \( b \)
(Assuming compiler doesn’t reorder variables in memory)

- Adds 1*\text{sizeof(char)} to the memory address
- Adds 1*\text{sizeof(int)} to the memory address

Pointer arithmetic should be used \text{cautiously}
A Special Pointer in C

Special constant pointer **NULL**

- Points to no data
- Dereferencing illegal – causes *segmentation fault*

- To define, include `<stdlib.h>` or `<stdio.h>`
Generic Pointers

void *: a “pointer to anything”

Lose all information about what type of thing is pointed to

- Reduces effectiveness of compiler’s type-checking
- Can’t use pointer arithmetic

type cast: tells the compiler to “change” an object’s type (for type checking purposes – does not modify the object in any way)

Dangerous! Sometimes necessary...

```c
void  *p;
int    i;
char   c;
p = &i;
p = &c;
putchar(*(char *)p);
```
Pass-by-Reference

```c
void set_x_and_y(int *x, int *y)
{
    *x = 1001;
    *y = 1002;
}

void f(void)
{
    int a = 1;
    int b = 2;

    set_x_and_y(&a, &b);
}
```
Arrays and Pointers

Dirty “secret”:
Array name ≈ a pointer to the initial (0th) array element

\[ a[i] \equiv *(a + i) \]

An array is passed to a function as a pointer

- The array size is lost!

Usually bad style to interchange arrays and pointers

- Avoid pointer arithmetic!

Passing arrays:

Really

\[
\text{int } \ast \text{array}
\]

Must explicitly pass the size

int
foo(int array[],
    unsigned int size)
{
    ... array[size - 1] ...
}

int
main(void)
{
    int a[10], b[5];
    ... foo(a, 10)... foo(b, 5) ...
}
Arrays and Pointers

```c
int foo(int array[],
       unsigned int size)
{
    ...
    printf("%d\n", sizeof(array));
}

int main(void)
{
    int a[10], b[5];
    ...
    foo(a, 10) ...
    foo(b, 5) ...
    printf("%d\n", sizeof(a));
}
```

What does this print?  8
... because `array` is really a pointer

What does this print?  40
Arrays and Pointers

```c
int i;
int array[10];

for (i = 0; i < 10; i++)
{
    array[i] = ...;
}

int *p;
int array[10];

for (p = array; p < &array[10]; p++)
{
    *p = ...;
}
```

These two blocks of code are functionally equivalent
Strings

In C, strings are just an array of characters
- Terminated with ‘\0’ character
- Arrays for bounded-length strings
- Pointer for constant strings (or unknown length)

```c
char str1[15] = "Hello, world!\n";
char *str2     = "Hello, world!\n";
```

C, ...

Pascal, Java, ...

C terminator: ’\0’
String length

Must calculate length:

```c
int strlen(char str[])
{
    int len = 0;
    while (str[len] != '\0')
        len++;
    return (len);
}
```

Provided by standard C library: ```#include <string.h>```
Pointer to Pointer (char **argv)

Passing arguments to main:

```c
int main(int argc, char **argv)
{
    ...
}
```

Suppose you run the program this way

UNIX% ./program hello 1 2 3

argc == 5 (five strings on the command line)
char **argv

argv[0] 0x1000
argv[1] 0x1008
argv[2] 0x1010
argv[3] 0x1018
argv[4] 0x1020

0x1000 argv[0] 0x1000
0x1008 argv[1] 0x1010
0x1010 argv[2] 0x1018
0x1018 argv[3] 0x1020
0x1020 argv[4]

“./program”
“1”
“hello”
“2”
“3”

These are strings!!
Not integers!
Next Time

Structures and Unions