

Arrays and Pointers in C

Mrs.P.Rupa Ezhil Arasi
Associate Professor/CSE
MEC (Autonomous)

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

```
int array[10];  
int b;
```

Compare: C: int array[10];
Java: int[] array = new int[10];

```
array[0] = 3;
```

First element (index 0)

```
array[9] = 4;
```

Last element (index size - 1)

```
array[10] = 5;
```

```
array[-1] = 6;
```

No bounds checking!

Allowed – usually causes no *obvious* error

array[10] may overwrite b

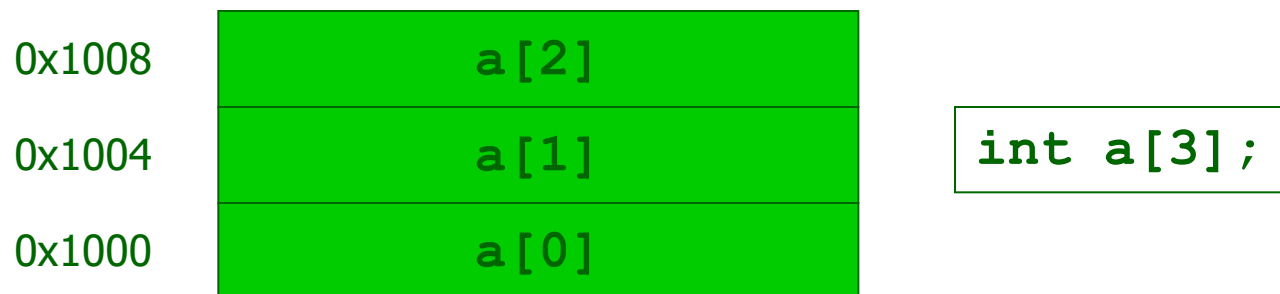
Array Representation

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, ...**

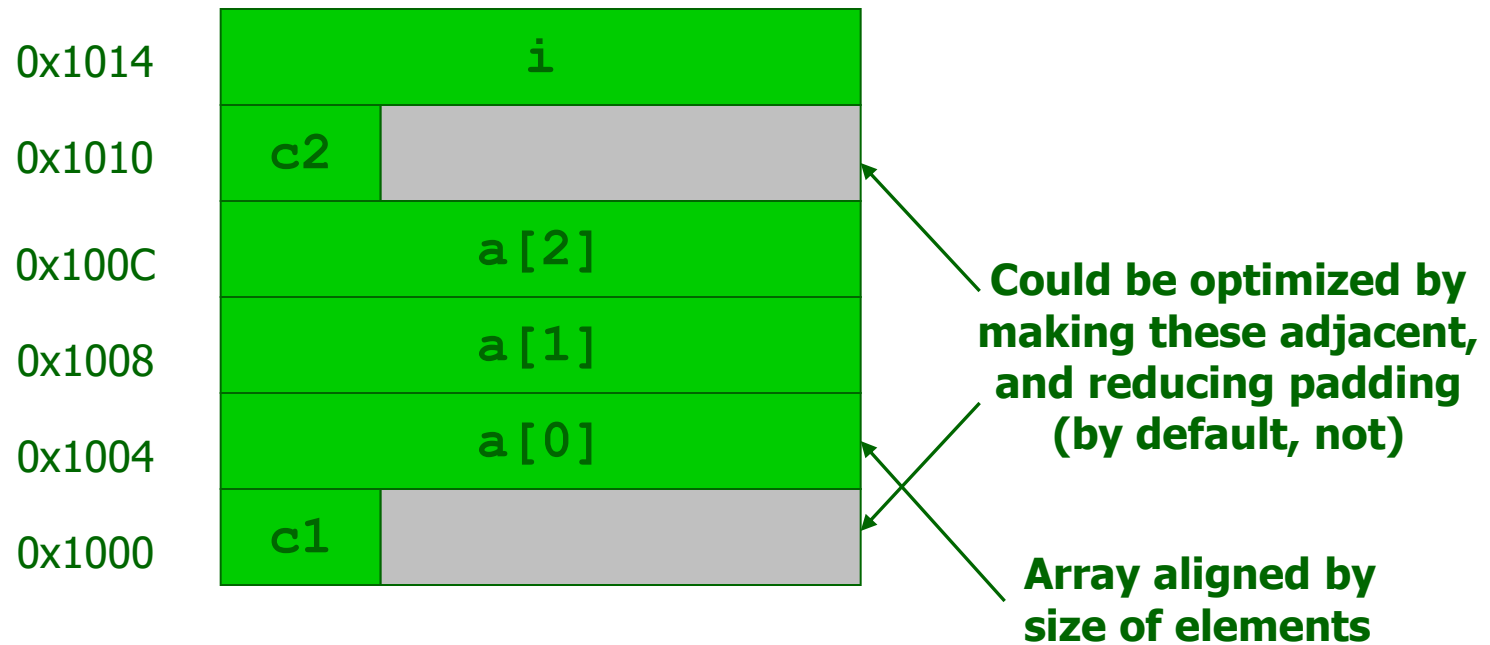
m and s are not part of representation

- ♦ **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**



Array Representation

```
char    c1;  
int     a[3];  
char    c2;  
int     i;
```



Array Sizes

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

What is

sizeof(array[3])? 4

returns the size of
an object in bytes

sizeof(array)? 40

Multi-Dimensional Arrays

```
int  matrix[2][3];  
  
matrix[1][0] = 17;
```

Recall: no bounds checking

What happens when you write:

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

0x1014

matrix[1][2]

0x1010

matrix[1][1]

0x100C

matrix[1][0]

0x1008

matrix[0][2]

0x1004

matrix[0][1]

0x1000

matrix[0][0]

**“Row Major”
Organization**

Variable-Length Arrays

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

New C99 feature: Variable-length arrays
defined within functions

Global arrays must still have fixed (constant) length

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

Pointers

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

```
int *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

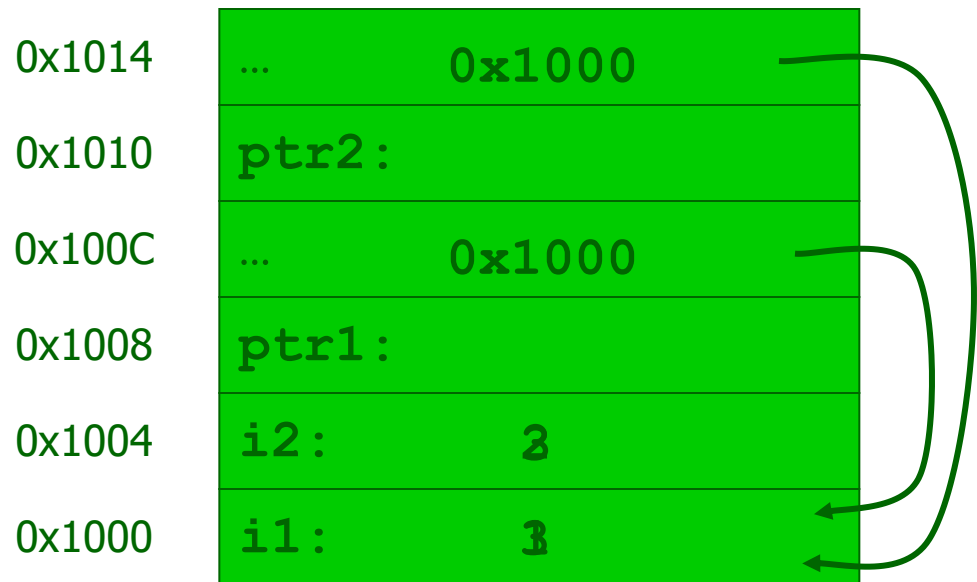
Using Pointers

```
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;
```

What happens?

Type check warning: int_ptr2 is not an int

int1 becomes 8

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 = int_ptr2;
```

What happens?

Type check warning: `*int_ptr2` is not an `int` *

Changes `int_ptr1` – doesn't change `int1`

Pointer Arithmetic

pointer + number

pointer - number

E.g., *pointer + 1* adds 1 something to a pointer



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

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

Pointer arithmetic should be used cautiously

A Special Pointer in C

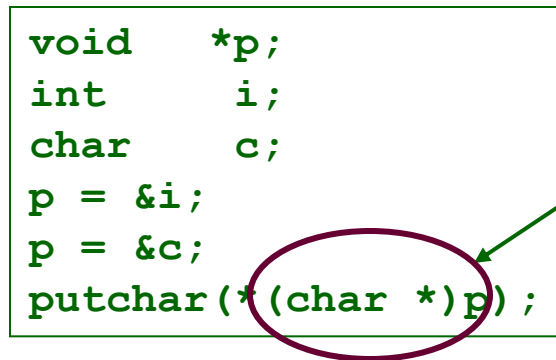
Special constant pointer `NULL`

- ◆ Points to no data
- ◆ Dereferencing illegal – causes *segmentation fault*
- ◆ To define, include `<stdlib.h>` or `<stdio.h>`

Generic Pointers

void *: a “pointer to anything”

```
void    *p;  
int     i;  
char    c;  
p = &i;  
p = &c;  
putchar(*(char *)p);
```



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...

Lose all information about what type of thing is pointed to

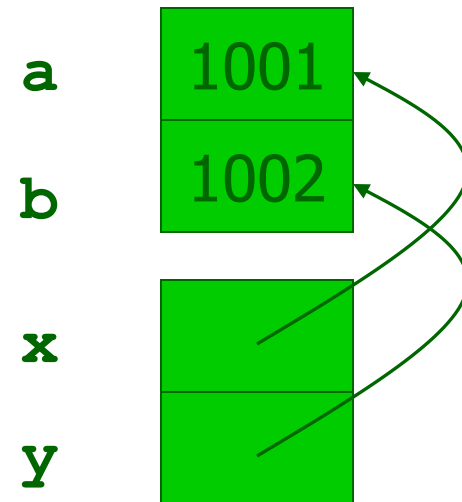
- ◆ Reduces effectiveness of compiler’s type-checking
- ◆ Can’t use pointer arithmetic

Pass-by-Reference

```
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 \approx 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 int *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

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

What does this print? **8**

... because `array` is really
a pointer

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

What does this print? **40**

Arrays and Pointers

```
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)

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

C, ...

H	e	l	l	o	,		w	o	r	l	d	!	\n	terminator
---	---	---	---	---	---	--	---	---	---	---	---	---	----	------------

C terminator: ‘\0’

Pascal, Java, ...

length	H	e	l	l	o	,		w	o	r	l	d	!	\n
--------	---	---	---	---	---	---	--	---	---	---	---	---	---	----

String length

Must calculate length:

```
int
strlen(char str[])
{
    int len = 0;

    while (str[len] != '\0')
        len++;

    return (len);
}
```

array access to pointer!

can pass an array or pointer

Check for terminator

What is the size of the array???

Provided by standard C library: #include <string.h>

Pointer to Pointer (char **argv)

Passing arguments to main:

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

size of the argv array/vector

an array/vector of char *

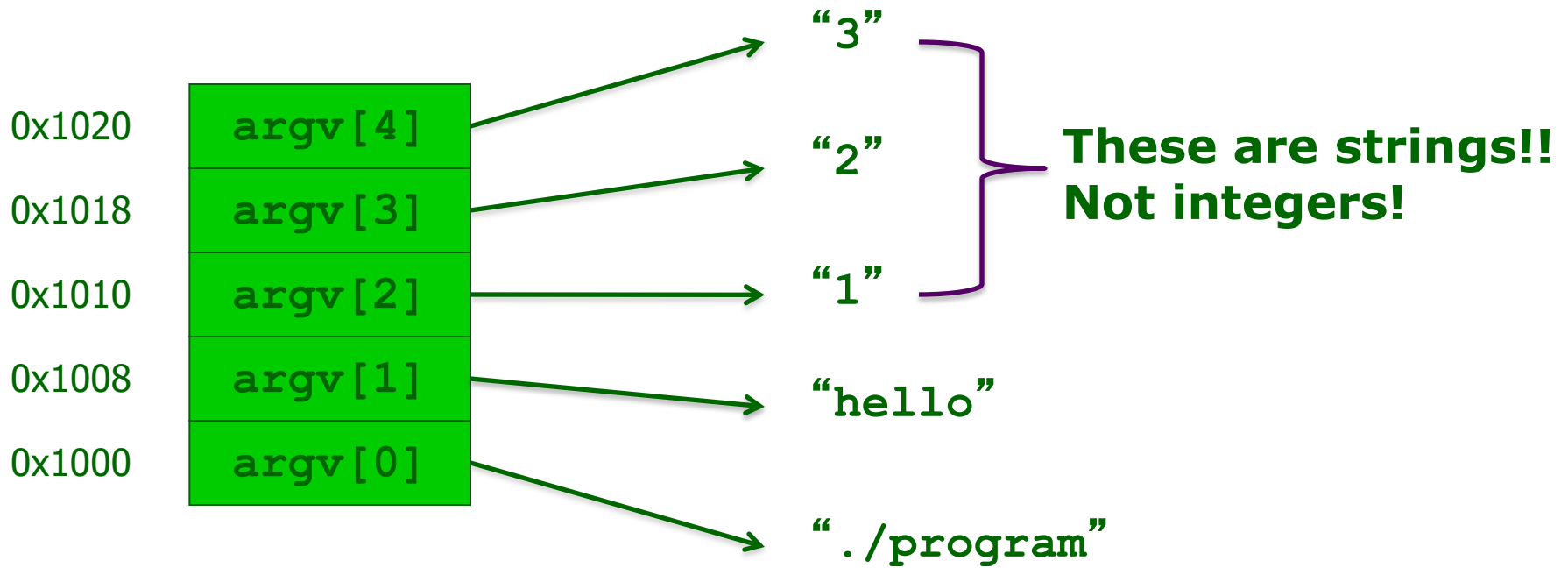
Recall when passing an array, a pointer to the first element is passed

Suppose you run the program this way

```
UNIX% ./program hello 1 2 3
```

argc == 5 (five strings on the command line)

char **argv



Next Time

Structures and Unions