Arrays & strings

Derrick

Arrays

An *Array* is a collection of data items (elements) all of the same type.

C always allocates the array in a single block of memory.

The *size* of the array, once declared, is *fixed* forever.

```
    ex) int arr[7];
```

Initializing Arrays

- The number of initializing values is exactly the same as the number of elements in the array. In this case the values are assigned one to one,
  - `int a[5] = { 1, 2, 3, 4, 5 };`
- The number of initializing values is less than the number of elements in the array. Here the values are assigned “one to one” until they run out. The remaining array elements are initialized to zero
  - `int a[5] = { 1, 2 };`
- The number of elements in the array has not been specified, but a number of initializing values has. Here the compiler fixes the size of the array to the number of initializing values and they are assigned one to one.
  - `int a[] = { 1, 2, 3, 4, 5 };`

Array Names

- The name of an array is a pointer to the start of the array. (the zeroth element)

```
int arr[2] = { 1,  2 };

int *ptr = &arr[0];

ptr == arr;
```

Cannot Assign to an Array

- Array names (variables) are ***constant.***

```
int a[10];

int b[10];

a = b; /* compile-error! */

/* array variables are constant (read-only) and thus
cannot be assigned to another value. */
```

Passing Arrays to Functions

- When an array is passed to a function a pointer to the zeroth element is passed across. (by reference)
- You need to provide the valid number of elements (size).

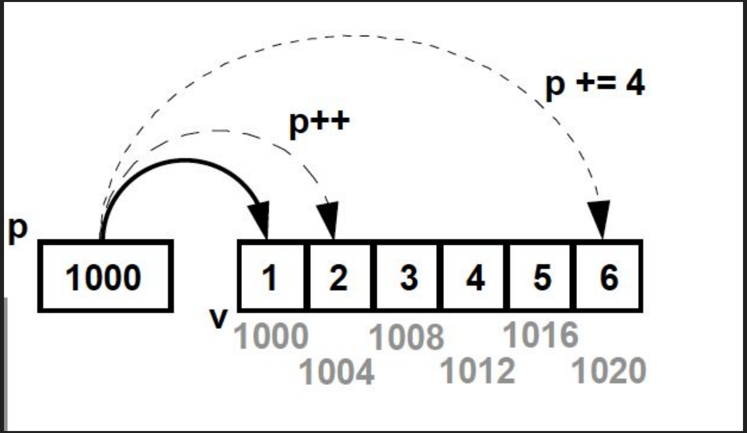
```
int add_elements(int a[], int size);

/* same */

int add_elements(int *a, int size);
```

Practice

- Create a function that takes an int array and returns the sum of the array.

Using pointers

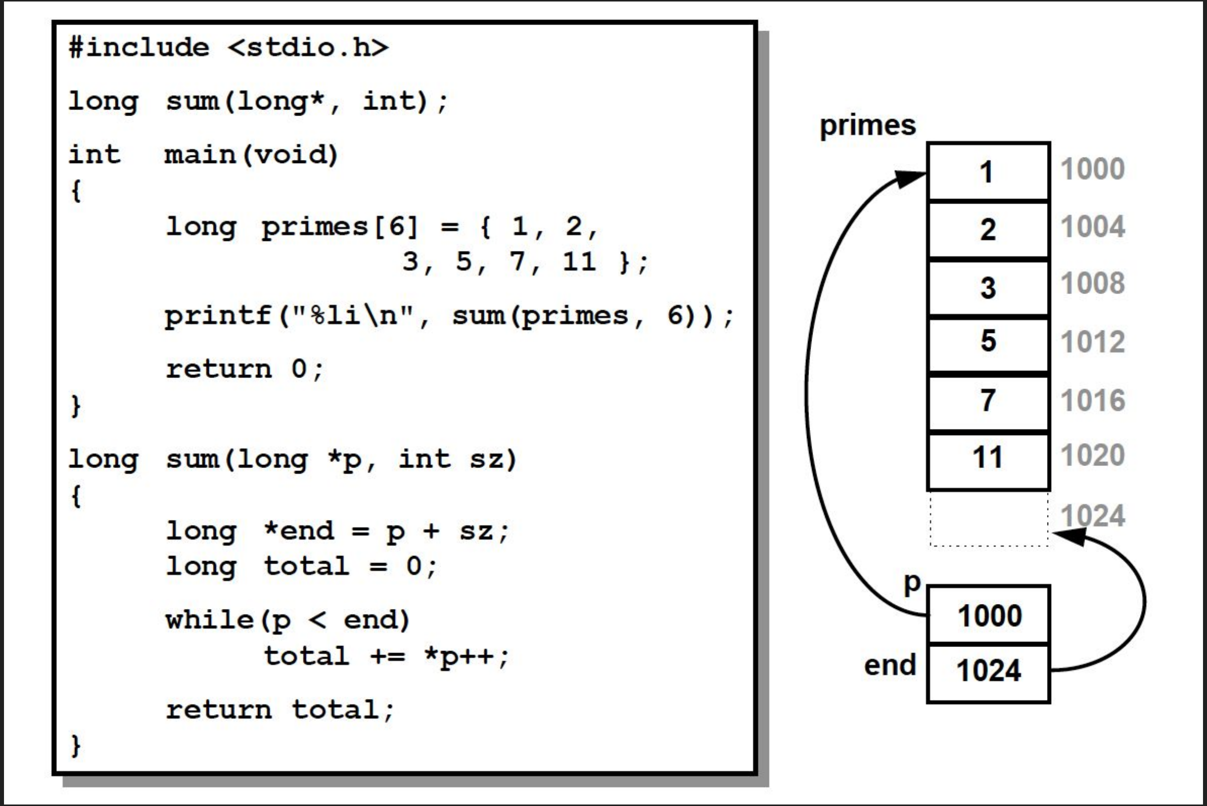
- You can use pointers to access array elements rather than using constructs involving “`[ ]`”
- Pointers in C are automatically scaled by the size of the object pointed to when involved in arithmetic.
- `int : 4 bytes, char : 1 byte, etc.`



Example

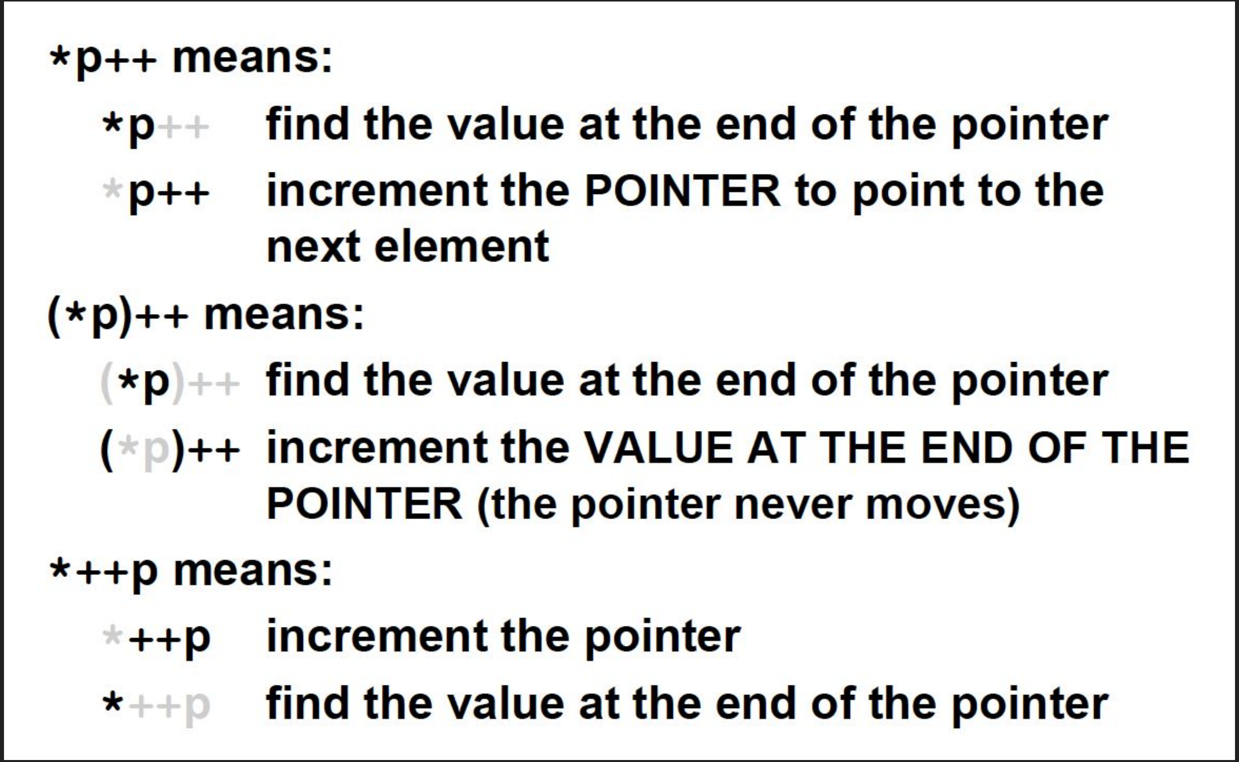
```
int arr[5] = { 1, 2, 3, 4, 5 };

int *ptr = arr;

ptr++;

ptr--;

ptr += 4;

int size = ptr - arr + 1; /* scaled */
```

Example



```
#include <stdio.h>

long  sum(long*, int);

int   main(void)
{
      long  primes[6] = { 1, 2,
                      3, 5, 7, 11 };

      printf("%li\n", sum(primes, 6));

      return 0;
}

long  sum(long *p, int sz)
{
      long  *end = p + sz;
      long  total = 0;

      while(p < end)
            total += *p++;

      return total;
}
```

++ and *



***p++ means:**

***p++**	**find the value at the end of the pointer**
***p++**	**increment the POINTER to point to the next element**

(*p)++ means:

(*p)++	**find the value at the end of the pointer**
(*p)++	**increment the VALUE AT THE END OF THE POINTER (the pointer never moves)**

***++p means:**

***++p**	**increment the pointer**
***++p**	**find the value at the end of the pointer**

Which Notation? `arr[i] vs *(arr + i)`

```
/* same */
arr[i] == *(arr + i)
```

Strings

- C has no native string type, instead we use array of `chars`
- A special character, called null character ‘`\0`’, marks the end.
- Depending on how arrays of chars are built, we may need to add the null by hand, or the compiler may add it for us.
- `NULL != '\0'   /* pointer != char */`
- `0 != '\0'    /* int != char */`

Example

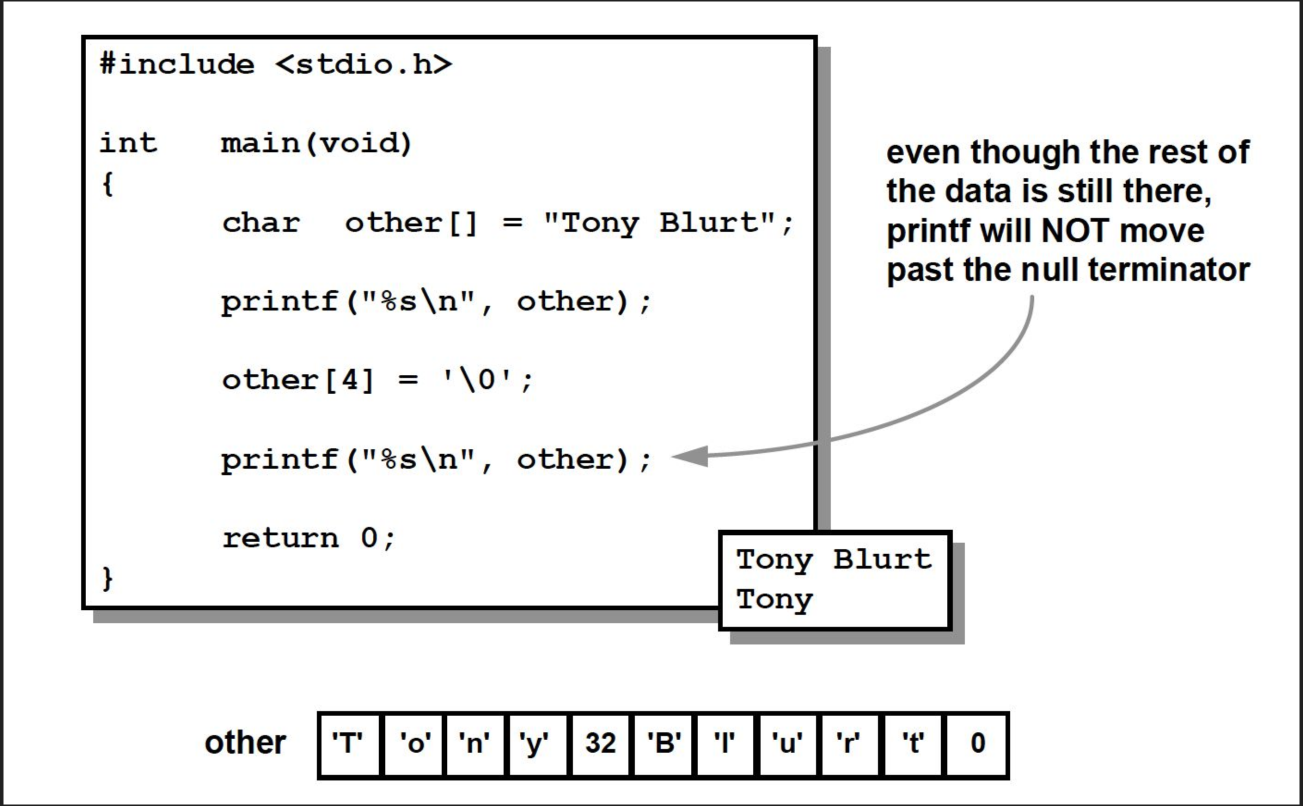
```
char first_name[8] = { ‘D’, ‘e’, ‘r’, ‘r’, ‘i’, ‘c’, ‘k’, ‘\0’ };

char first_name[8] = “Derrick”;

char first_name[] = “Derrick”;

char *first_name = “Derrick”;

char arr_char[7] = “Derrick”;  /* array of chars */
```

Printing strings

- `int puts(const char *)` to print strings.
  - `#include <stdio.h>`
- `printf` supports `“%s”` format specifier for strings.

- Assignment 2: `void vc_putstr(char *str)`

Null really Does mark the end



```
#include <stdio.h>

int    main(void)
{
       char   other[] = "Tony Blurt";

       printf("%s\n", other);

       other[4] = '\0';

       printf("%s\n", other);

       return 0;
}
```

even though the rest of the data is still there, printf will NOT move past the null terminator

```
Tony Blurt
Tony
```

other										
'T'	'o'	'n'	'y'	32	'B'	'l'	'u'	'r'	't'	0

Assigning to Strings

- Strings can be initialized with =, but not assigned with =
- Remember the name of an array is a CONSTANT pointer to the zeroth element.

```
char who[] = “Lukaku”;

who = “Yukako”; /* nope */

char *country = “Belgium”;

country = “Japan”; /* yes */
```

strcpy

- Assigning a string. (copy)
- `#include <string.h>`
- `char *strcpy(char *dest, const char *src);`
- `strcpy(who, “Yukako”); /* yes */`
- How does it copy?
- `strcpy` does absolutely no bounds checking.

strncpy

- `strcpy(who, “a really long string instead”);`

- Overflow the array “who” and corrupt the memory around it. (might crash!)

- Use `char *strncpy(char *dst, const char *src,` **`size_t`** `len);`

- `size_t == unsigned long` (use for sizes or counting)

- `strncpy(who, “a really long string instead”, sizeof(who));`

- Unfortunately when strncpy hits the count first, it fails to null terminate. We have to do this by hand:

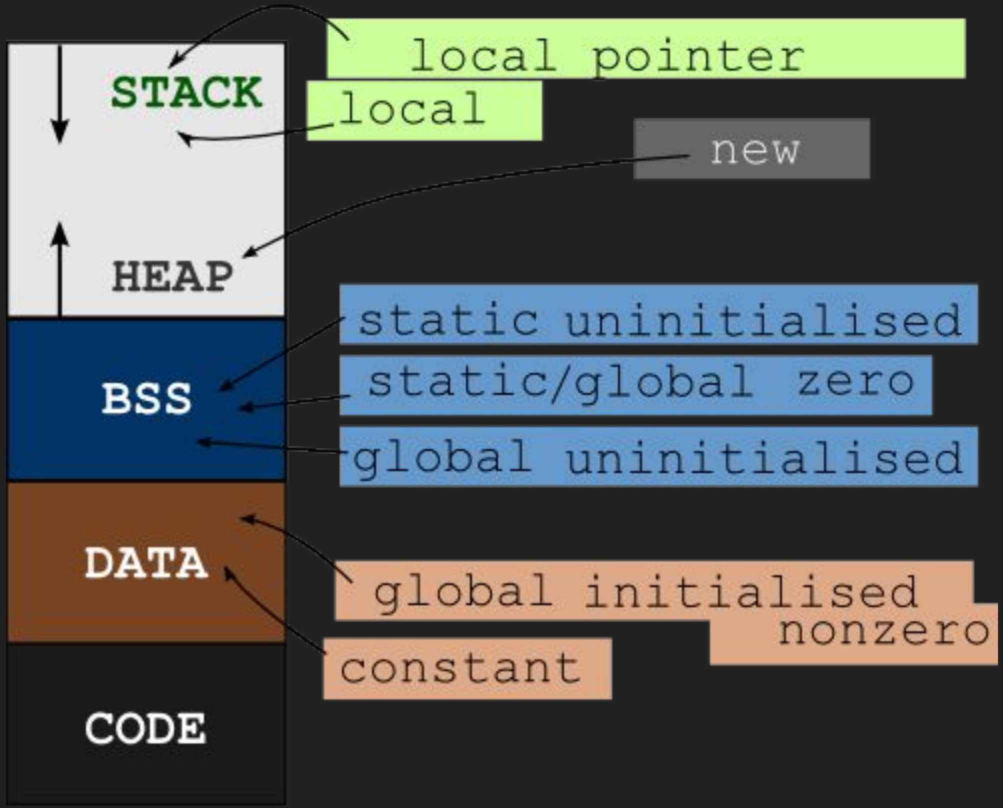
  `who[sizeof(who) - 1] = ‘\0’;`

Pointing to Strings

- To save us declaring many character arrays to store strings, the compiler can store them directly in the data segment.
- The compiler may recycle some of these strings, therefore we must NOT alter any of the characters.

```
char str[] = “This is a string”; (char array - stack)

char *str = “This is a string”; (immutable str - data)



=> const char *str = “Data segment”;
```

Memory Layout (segmentations) **(optional)**



Multi-dimensional Arrays in C

- `int threedim[5][10][4];`
- `int twodim[3][3] = { {0, 1, 2}, {3, 4, 5}, {6, 7, 8} };`
- `int twodim[3][3] = { 0, 1, 2, 3, 4, 5, 6, 7, 8 };`

- *Do not declare with pointers, but you can access using pointers.*