# Sorting, Searching and Strings in C

Theory is when you know something but it doesn't work. Practice is when something works but you don't know why. Programmers combine theory and practice: nothing works and they don't know why.

# String Handling

- string manipulation is very common in programming problems
- C does not have built-in support for strings, even the format (null character terminated) is a convention
- the standard C library has many functions that help with string handling

#include <string.h>

# String Functions - simple

- Copying

strcpy(char * dest, char * src)

Note: ‘dest’ must point to an area of memory big enough to take the characters from ‘src’ plus the terminating null character

# String Functions - simple

- Copying with a limit: to be safe we can put a limit on the number of characters copied

strncpy(char * dest, char * src, int count)

**strncpy – good practice**
**strcpy – convenient, but trusts programmer**

# String Functions - simple

- Concatenating

strcat(char * dest, char * src)

Note: ‘dest’ must point to an area of memory big enough to take the characters from ‘dest’ **plus the characters from ‘src’** plus the terminating null character

# String Functions - simple

- Concatenating can also be done with a limit:

strncat(char * dest, char * src, int count)

**strncat – good practice**
**strcat – convenient, but trusts programmer**

# String Functions - simple

- Comparing

int strcmp(char * s1, char * s2)

returns a number greater, less or equal to zero if s1 is greater, less or equal to s2

int strncmp(char * dest, char * src, int count)

# String Functions - more complex

- searching for substring:

```
char * strstr (const char *haystack,
              const char *needle)
```

find first occurrence of string “needle” in string “haystack”

# Regular Expressions

- regular expressions are patterns that can be matched against strings

eg “[Hh]ello$” will match “Hello” or “hello” at the end of a line

# Regular Expressions

- can be very powerful but very complex
- they are another “little language” that you have to debug
- often implemented inefficiently
- if someone has a difficult problem and says “I know, I’ll use regular expressions!” then they have two difficult problems…..

# C Functions for Regular Expressions

- regular expression searching:

char *regcmp(const char *pat)

compile “pat” into executable form

char *regex(const char *re, const char *subject)

match compiled regular expression “re” against string “subject”

#include <regex.h>

# C Functions for Regular Expressions

- regular expression searching:

int regcomp(regex_t *preg, const char *regex, int cflags);

compile into executable form, saved to preg

int regexec(const regex_t *preg, const char *string, size_t nmatch, regmatch_t pmatch[], int eflags);

match compiled regular expression preg against string “string”, nmatch/pmatch for location information

#include <regex.h>

# Formatting functions

- the function sprintf is like printf except the output goes to a string instead of a file

eg sprintf(outstr, “Answer is %d\n”, num);

this will format the number and put the result into the string “outstr”

# Formatting functions

- the function “sscanf” is like “scanf” except the string to be scanned is from memory rather than from a file.

eg sscanf(instr, “%d, %d”, &num1, &num2);

the string “instr” is scanned for 2 integers.

# Useful code

- you can use sprintf to create formatted strings for many purposes
- on input you can read a line (eg using getline) and then rescan it as many times as you want using sscanf. You can even use sscanf on part of a line.

# Sorting and Searching

- sorting and searching are very common operations on data
- there are well known algorithms for sorting and searching
- you could implement your own function
- the C library contains efficient sort and search functions

# Sorting

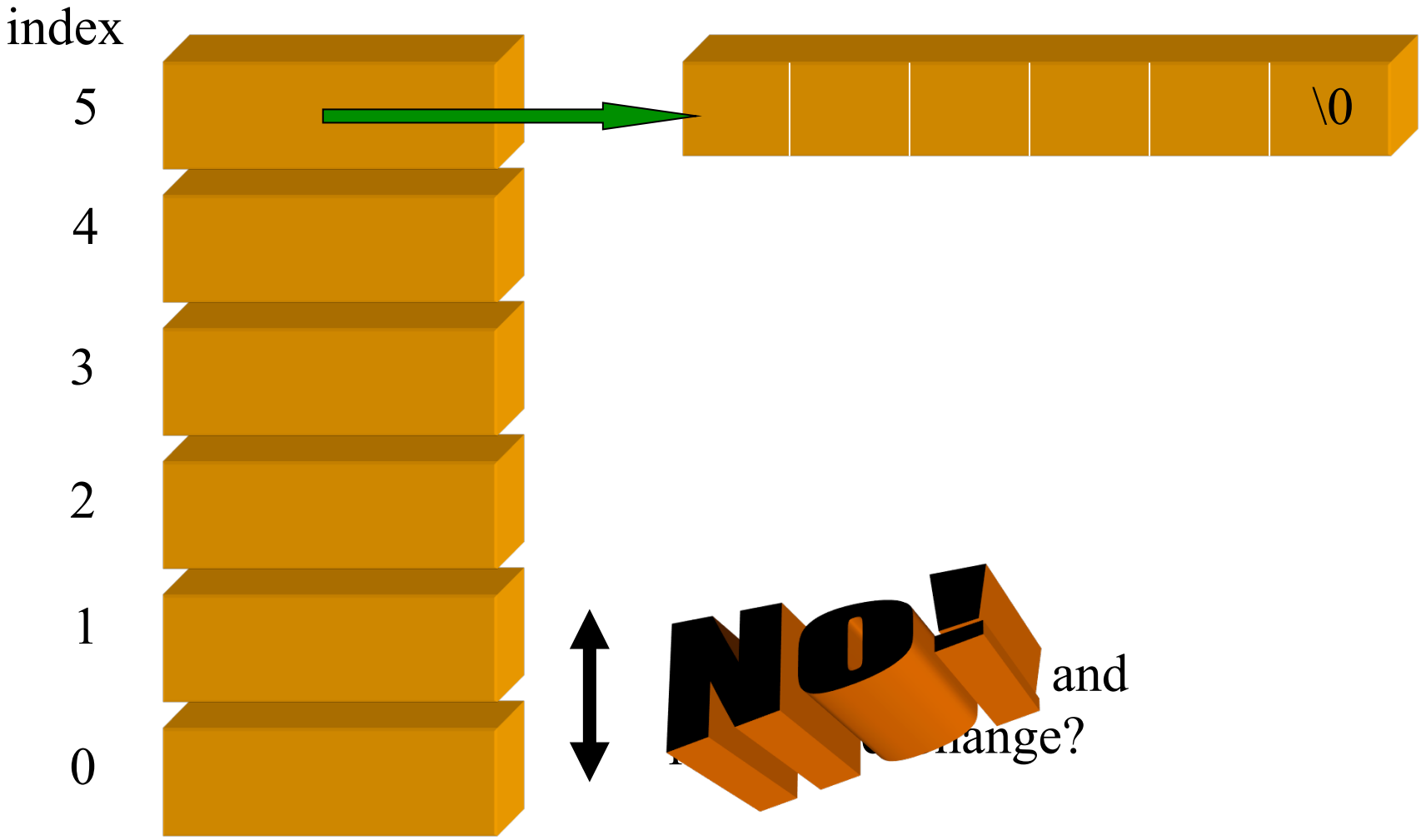
- many algorithms - study them in a later course
- simple algorithm: “bubble sort”
- more sophisticated/complex (and faster): quicksort, shell sort,….

# Sorting strings

- C strings are represented as a pointer to an array of characters terminated by a null character ( ‘\0’ )
- an array of strings is an array of pointers
- how do we sort an array of strings?

# Sorting Strings

array of pointers to strings




and

hange?

# Comparison Function

```
/*
** returns -1 if s1 < s2
**          0 if s1 == s2
**          1 if s1 > s2
** (warning - untested code!)
*/
int
compare(char *s1, char *s2)
{
        for (; (*s1 != '\0')&&(*s2 != '\0'); )
                if (*s1 < *s2)
                        return -1;
                else if (*s1 > *s2)
                        return 1;
                else
                {
                        s1++;    s2++;
                }
        if ((*s1 == '\0')&&(*s2 == '\0'))
                return 0;
        if (*s1 == '\0')
                return -1;
        else
                return 1;
}
```

## Comparison Function using C library function

```
/*
** returns -1 if s1 < s2
**          0 if s1 == s2
**          1 if s1 > s2
*/
int
compare(char *s1, char *s2)
{
        return strcmp(s1, s2);
}
```

# Sorting

- how can we write a *general purpose* sorting function: one that can sort arrays of any object (eg integers, strings, structs …)?
- need to change the comparison function for every different data type
- need to somehow pass the comparison function to our general sort function
- use a *function pointer*

# C library function: qsort

- high performance algorithm
- extensively tested over many years
- no need to ever write your own sort function
  - use qsort

```
#include <stdlib.h>
void
qsort(void *base, int nelements, int eltsize,
        int (*compare)(const void *, const void *))
```

- “compare” is the name of the comparison function declared elsewhere

# qsort example

- sorting an array of strings

```
char * strings[100];

qsort(strings, 100, sizeof(char *), compare)
```

- compare function shown earlier

# Sorting structures

- reading a file of records (structs), sorting them and writing them out

# Searching

- like sorting, searching is a very common operation that should be done efficiently
- searching also requires a comparison function
- the C library has a seach function called *bsearch* that implements the *binary search algorithm*
- binary search assumes the array to be searched is in sorted order (how do we do that? :-)
- binary search operates by looking at the middle element of the array to be searched and determining if the element we're looking for is in the first or second half and then repeating the process

# Searching

- bsearch needs a pointer to an element you are searching for as well as a pointer to the array, the number of elements, the size of each element and a pointer to a comparison function
- bsearch returns a pointer to the element it found or NULL if not found

```
void
*bsearch(const void *key, const void *base,
        int number, int size,
        int(*compar)(const void *, const void *))
```

# Function Pointers

- in each case (qsort, bsearch) the comparison function is passed to the library function using a *function pointer*
- the declaration of the function pointer parameter looks like:

```
type (*f)(param declaration…)
```

- and the call of the function looks like:

```
f(params…)
```

end of segment