CS205

C / C++

Stéphane Faroult
faroult@sustc.edu.cn

Wang Wei (Vivian) vivian2017@aliyun.com

I'd like to review a few points that take C apart from some other languages like Java. Loops, conditions, functions are features you find in almost every programming language. C is really special in the way it lets you handle data in your program.

A review of what is **REALLY** important

7 POINTS

This is true of any language, but C gives you more choices.

1 Whatever you do in a computer, you need to reserve bytes to store your data

declaration → data area of memory
declaration → stack *Limited lifespan*

dynamic allocation heap *Must free!*

If you ask the system to give you memory, you need a place to store the address, and you have to declare a pointer.

2 Dynamic allocation requires declaring pointers

declaration

```
STRUCT_T *strp  = NULL;
```

dynamic allocation

```
strp  = (STRUCT_T *)malloc(sizeof(STRUCT_T));
```

This is significantly different from Java; in Java, an `int` is never a reference (you need to use the object version, `Integer`, if you want a reference) and an object is always a reference.

3

ANYTHING can be declared either way

declaration

```
int      my_int = 0;
STRUCT_T struct_var;
```

≠

dynamic allocation

```
int      *intp = (int *)malloc(sizeof(int));
STRUCT_T  *strp = (STRUCT_T *)malloc(
                       sizeof(STRUCT_T));
```

Don't confuse, especially with the char type, arrays and arrays of pointers.

4

Arrays are memory + pointer

```
int   int_array[5];
char  str[10];
char *strarr[5];
```

int_array

str

strarr

"string 1"

"string 2"

Constants, declared char arrays or malloc'ed

5

You can mix arrays and pointers

This is confusing when you start with C, but you can use with a memory chunk an array syntax, and the reverse is true as well.

```
int  *int_array = (int *)malloc(sizeof(int) * 100);
int   i;

for (i = 0; i < 100; i++) {
  int_array[i] = i;
}
```

```
int   int_array[100];
int   i;
int  *p = int_array;
for (i = 0; i < 100; i++) {
  *p = i;
  p++;      } Can also be written as *p++ = i;
}
```

When you call a function, you copy arguments in the stack. If you modify the argument in the function, it will not affect the caller.

6

You copy values on the stack when calling functions

It's more common to pass addresses of structures, because you copy fewer bytes and it's faster.

Simple variable

Full structure

Address (pointer, array)

But if a function gets a pointer, it can dereference the pointer and modify what is at this address – it will affect the caller.

To modify a parameter, you need to pass its address

It's the only option.

Example: scanf() (printf() doesn't need addresses)

You can say that a pointer is "const" if the function doesn't modify it

```
int printf(const char *fmt, ...)
```

A few hints about Lab2

I'd like to give you a few hints about Lab2, because this is really about program design. It's not too difficult to write a program that works in most cases; but the difference between junior and senior programmers isn't so much, usually, in algorithms, but more in the approach, and having something that is flexible enough to accommodate even the unexpected. Think "Zen mastery" ...

```
101116,"South University-Montgomery","5355 Vaughn Rd","Montgomery",
"AL","36116", 1, 5,"Victor Biebighauser","President","3343958800",
" ","261569113","01303906",1,"southuniversity.edu",
"southuniversity.edu","southuniversity.edu","southuniversity.edu",
"tcc.noellevitz.com/edmc/Transfer-Students?
iframe=true&width=600&height=1000",3,1,3,7,1,1,20,1,2,-2,2,2,12,1,
"A ",-2,-2,"-2",1,1,1,1,1," ",2,23,2,11,5,3,6,40,2,1,
33860,1,-2,-2,1,"Education Management Corporation","301790",
1101,"Montgomery County",102,-86.216488,32.342684
101143,"Enterprise State Community College",
"600 Plaza Drive","Enterprise","AL","36330-1300", 1, 5,
"Nancy Chandler, Ed.D.","President","3343472623","3343936223",
"630504851","00101500",1,"www.escc.edu","www.escc.edu",
"www.escc.edu","https://grace.escc.edu/cgi-bin/admonline.mbr/
login","www.escc.edu/NetPrice/npcalc.htm",
4,2,1,3,1,2,40,1,2,2,2,2,32,1,"A ",-2,-2,"-2",1,1,1,1,1,
"Enterprise-Ozark Community College  |Enterprise State Junior
College",4,2,1,-1,3,1,3,40,2,2,21460,2,222,-2,1,
"Alabama Community College System","101030",1031,"Coffee County",
102,-85.836956,31.297496
```

Ouch!

A comma indicates a new field

UNLESS between quotes

If you misinterpret a comma, you will think that you are in field n+1 when you are still in field n, and all the following fields will be shifted by one position. The presence of commas in fields is something that you cannot think about before having experienced it, and only some rows have the problem, so it may be difficult to spot. But it may break a program that tries to load your output.

The first idea of most of you is probably to read line by line (this is what you would do in Java). You may wonder which size you should give to you array. ?

First approach

And it's easier to read from stdin than to open a file.

```
char line[SOME_SIZE];

while (fgets(line, SOME_SIZE, stdin)) {
    // Process
}
```

There are some magical Unix commands. This one displays the length of all lines from the shortest to the longest one.

```
awk '{print length($0);}' unidata.csv | sort
```

But you need no magic (by the way, the longest line in the data provided is a little under 1000 bytes).

First approach

```
#define SOME_SIZE 5000

char line[SOME_SIZE];
```

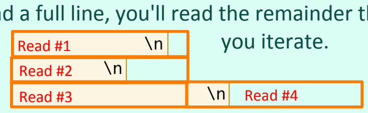
We are no longer in the 1970s, memory is cheap, most computers have plenty of it, and you can have an oversized array of 5K or 10K, which is still very small by modern standards (on the other hand many applications also use far more memory than they should...)

Not a real problem ...

First approach

```
#define SOME_SIZE 500

char line[SOME_SIZE];
```

But even if your array is far too small, if you use a safe function such as fgets() you will read up to \n or to the size of you array, whatever occurs first. If you cannot read a full line, you'll read the remainder the next time you iterate.



First approach

```
#define SOME_SIZE 500

char line[SOME_SIZE];
```

If you have read SOME_SIZE bytes and the last character is not \n, the next line will continue the present line and perhaps the current field. It's quite manageable.

If `line[strlen(line)-1]` is NOT '\n' we haven't read the line in full

Next line read continues the previous one

First approach

Real problem: splitting the line

~~strtok()~~

"Nancy Chandler, Ed.D."

The real problem is splitting the line, because strtok() will work on most lines, but not all of them.

First approach

If you are between quotes, you must simply not look at separators. So you really need to inspect characters one by one.

Real problem: splitting the line

```
loop on characters
    if not between quotes
        if separator then increase field number
        else if quote then set between quotes
    else
        if quote then unset between quotes
```

Second approach

But then if we process character by character, why bother with incomplete lines or safe line size? We can read from the input character by character, and when we get a \n we know that it's a new line. This is a typical Unix and C way.

Read by character, not by line

```
while character is not EOF
    if not between quotes
        if separator then increase field number
        else if quote then set between quotes
        else if '\n' start again from field one
    else
        if quote then unset between quotes
```

Second approach

Read by character, not by line

Read from stdin (getchar())

Write to stdout (putchar())

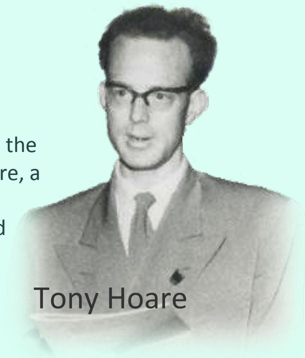
If a field was between quote, it would be safer to output it between quotes.

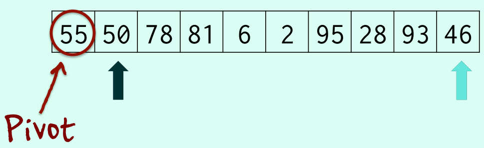
So in the end we can have a program that is both simpler (what is simpler than processing a single byte?) and able to handle almost any input. But simple ideas are rarely the first ones we have.

Revisiting recursion

Quick Sort

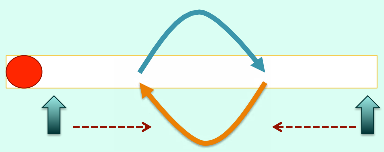
One of the first very good algorithms was invented in the early 1960s by Antony Hoare, a (then) young British mathematician, and named quick-sort.





There are several brilliant ideas in the algorithm. One of them is, instead of successively looking for smallest (or greatest) values, to take arbitrarily one value called pivot and find its final location.

QuickSort



To do so, we check each value by moving up and down in the array at once, and stopping when the "up" pointer encounters a greater value than the pivot, and the "down" pointer a smaller one. Those values are swapped.

55	50	46	28	6	2	95	81	93	78

When both pointers meet, everything on the right is bigger than the pivot, everything on the left is smaller (or equal) and we know that the final pivot location will be where the small red arrow points.

Smaller (or equal) Bigger

2	50	46	28	6	55	95	81	93	78

By swapping the pivot and the value occupying "its" place, we partition the original set into a subset containing smaller values, and a subset containing greater values.

The **BIG** idea

N^2 Sorting twice the number of values is 4 times as costly

Most simple sorting algorithms such as the famous bubble sort have a cost in number of operations (and time) that increases as the square of the number of values sorted. So it's better to perform two sorts applied to N values each than one sort applied to 2N values.

```c
int place_pivot(int *arr, int elem_count) {
    int pivot;
    int tmp;
    int up = 1;
    int down = elem_count - 1;

    pivot = arr[0];
    while (down > up) {
       while ((arr[up] <= pivot)
              && (up < down)) {
          up++;
       }
       while ((arr[down] > pivot)
              && (up < down)) {
          down--;
       }
```

Here is how the algorithm for placing the pivot can be coded in C.

```
        if (up < down) {
            // Exchange values
            tmp = arr[up];
            arr[up] = arr[down];
            arr[down] = tmp;
        }
    }
    // up has stopped at a value > pivot
    // or when it met the down pointer
    if (pivot < arr[up]) {
        // Place pivot at up - 1
        up--;
    }
    arr[0] = arr[up];
    arr[up] = pivot;
    return up;
}
```

The problem is that we must defer operations, sorting one subset, then the other, and if we apply each time the same "recipe" it can become very complicated.

$$\underbrace{2 \quad 50 \quad 46 \quad 28 \quad 6}_{\text{Sort}} \quad 55 \quad \underbrace{95 \quad 81 \quad 93 \quad 78}_{\text{Sort}}$$

2	50	46	28	6	55	95	81	93	78
0	1	2	3	4	5	6	7	8	9

"Remember" that we must sort 0 to 4

Sort 6 to 9

"Remember" that we must sort x to y

Sort w to z

"Remember" ...

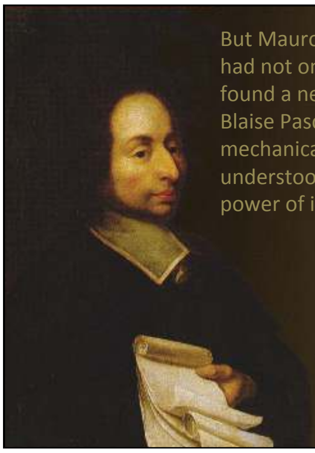
Mathematical Parenthesis

Thankfully, maths come to the rescue; not maths themselves, but a mathematical method which is very closely related to what we'll do.

MATHEMATICAL INDUCTION

This closely related mathematical method is induction, a very clever way of proving theorems.

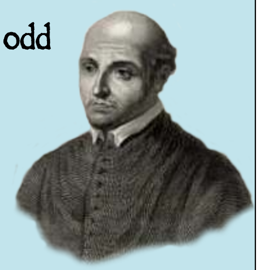
The first one to have used it (in 1575) is Francesco Maurolico, an Italian professor at the University of Messina.



But Maurolico didn't quite realize he had not only proved a theorem, but found a new way of reasoning. It's Blaise Pascal (of, among other things, mechanical computer fame) who fully understood in the next century the power of induction.

Blaise Pascal
(1623-1662)

The sum of the n first odd integers is equal to n^2.



This is what Maurolico proved, and to do it he used a two step method:

- Obvious for 1
- If it's true for N, it's true for N+1

Therefore it's true for any integer value.

Obvious for 0 and 1.

Suppose true for n.

$$1 + 3 + \dots + (2n - 1) = n^2$$

The sum of the (n+1) first odd integers is

$$\underbrace{1 + 3 + \dots + (2n - 1)}_{n^2} + (2n + 1)$$

$$n^2 + (2n + 1) = (n + 1)^2$$

It means "this is what had to be proved"; often shortened to QED in modern English (it's Latin) → **Quod Erat Demonstrandum**

Mathematical induction is based on these two elements:

Link between *n* and *n + 1*

Trivial case

Related (although not identical) thought in programming:

RECURSION

Mathematical Induction

Link between *n* and *n + 1*

⇧

Trivial case

Mathematical induction goes from the trivial case towards infinity.

Recursion

Link between *n + 1* and *n*

⇩

Trivial case

Recursion, which we'll use for the Quick-Sort, works by identifying a trivial case, then by assuming that we can solve a problem at level n-1 and expressing the solution to the problem at level n as a function of the n-1 level solution.

Recursion

A function contains a call to itself (with other parameters).

Recursion is characterized by functions that contain calls to themselves, with different parameters corresponding to a smaller problem.

~~**re·cur·sive** *adjective* \ri-ˈkər-siv\ *See* recursive~~

Not the way it works

Of course at one point the function must reach a very easy case and no longer call itself! There will necessarily be a condition in the function to stop the recursion.

With a recursive function you MUST first identify trivial cases.

Sort Sort

2	50	46	28	6	55	95	81	93	78
0	1	2	3	4	5	6	7	8	9

```
void quicksort(int *arr, int elem_count)
```

Trivial case ?

Cases 0 or 1 are kind of easy ...

Recursion: QuickSort

Smaller Bigger

When you have 0, 1 or even 2 values to sort, it's not much of a problem. Now assume than you have many values to sort. Place your pivot. Assume that you can sort a smaller number of values. Then you can sort the subset of smaller values. You can also sort the subset of bigger values. Then everything is sorted. Magic.

```c
void quicksort(int *arr, int elem_count) {
    int limit;
    int tmp;

    switch (elem_count) {
       case 0:
       case 1: // Do nothing
            break;
       case 2:
            if (arr[1] < arr[0]) {
               tmp = arr[1];
               arr[1] = arr[0];
               arr[0] = tmp;
            }
            break;
```

Here is what a rough version (can be improved) might look like.

These are the trivial cases. No call of anything.

Non trivial case. Assume that the function we are writing works for a smaller number of elements. Place the pivot (using the previously seen function) and return the index in the array just before it.

```c
        default:
          limit = place_pivot(arr, elem_count);
          // Now recursion
          quicksort(arr, limit);
          quicksort(&(arr[limit+1]),elem_count-limit-1);
          break;
     }
}
```

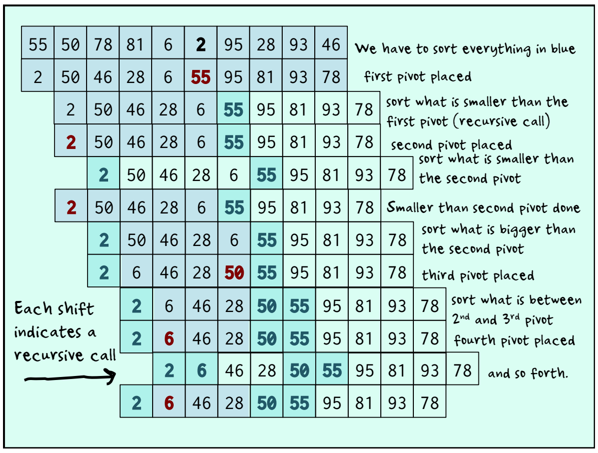
Then call the function for sorting what's on the left of the pivot (smaller elements), then what is on the right (bigger elements). Done. Very easy to write.

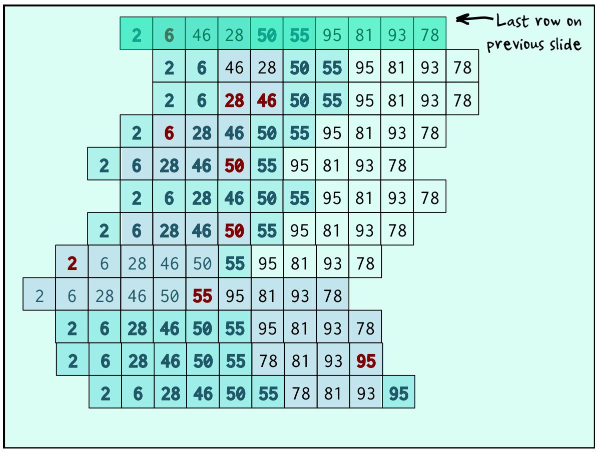
The algorithm can be improved (clever choice of pivot, sorting what is smaller first, etc ...).
Can also avoid calling functions for array size less than 2 ...

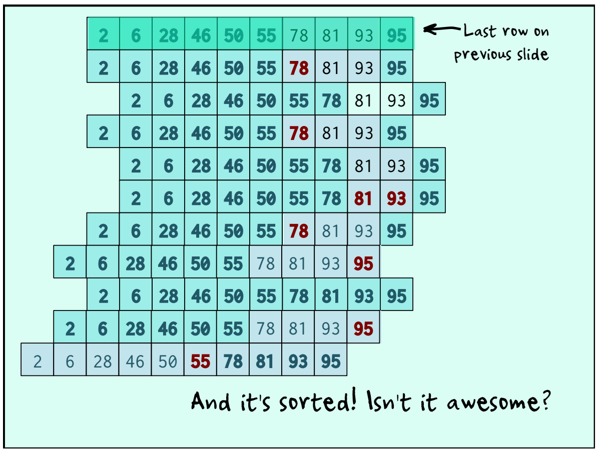
Why it works.

Still the magic of the stack.

Because of the stack mechanism, operations that have to be performed accumulate in the stack, and are popped out of the stack when done. It's the stack that keeps track of everything. What occurs is usually fairly complicated, but not the program.


We have to sort everything in blue
first pivot placed
sort what is smaller than the first pivot (recursive call)
second pivot placed
sort what is smaller than the second pivot
Smaller than second pivot done
sort what is bigger than the second pivot
third pivot placed
sort what is between 2nd and 3rd pivot
fourth pivot placed
and so forth.
Each shift indicates a recursive call


Last row on previous slide


Last row on previous slide
And it's sorted! Isn't it awesome?


Flickr: Mad Wraith

Execution is horribly complicated ...

...but writing is easy.

DON'T use recursion

where loops are easy to write.

Even if recursion can be an easy, and elegant, way of solving hairy problems, the big benefit of recursion is when operations to perform multiply out of control - the quick sort is a good example because everytime you want to sort a set, you end up with two smaller sets to sort.

If you know the sorcerer's apprentice in Disney's "Fantasia" ...

Traditional (stupid IMHO) recursion example:

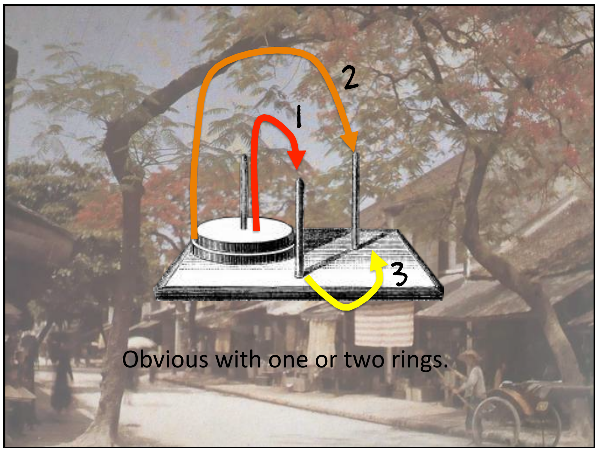
As a factorial can be defined as
n! = n x (n-1)!
it is often used as an example for teaching recursion.

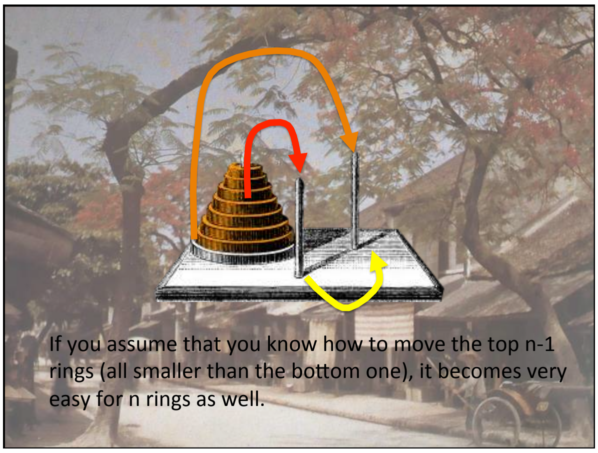
```
long fact(int n) {
   if (n == 0) {
      return (long)1;      Trivial case
   } else {
      return n * fact(n - 1);Recursion
   }
}
```

Why not loop ?

Unless you have already pre-computed a number of factorial values and don't need to go all the way down to 1, a loop is not more complicated and is more efficient (doing things in the stack has a cost)

```
long fact(int n) {
   long result = 1;
   int  i;

   for (i = 1; i <= n; i++) {
      result *= i;
   }
   return result;
}
```



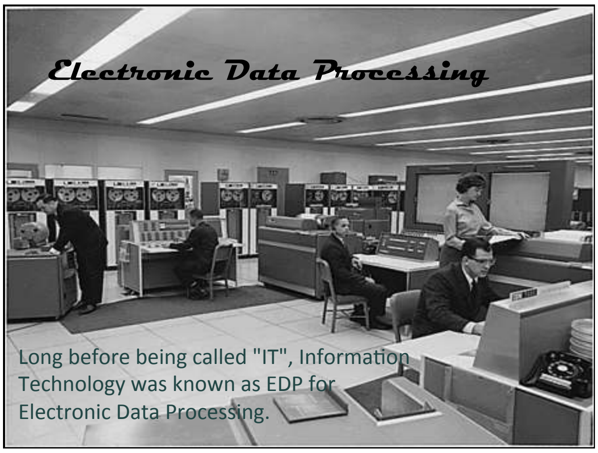



```
move_tower(tower_size, from_peg, to_peg, using_peg)
  if tower_size is 1
      move ring from from_peg to to_peg
  else  if tower_size is 2
        move top ring from from_peg to using_peg
        move bottom ring from from_peg to to_peg
        move top_ring from using_peg to to_peg      Note!
  else
      move_tower(tower_size -1, from_peg, using_peg, to_peg)
      move bottom ring from from_peg to to_peg
      move_tower(tower_size - 1, using_peg, to_peg, from_peg)
```

Data Structures

Collections

Let's now see how we deal in C with data structures, which are the equivalent of Java collections (except, as usual, that Java hides from you a lot of what is going on while C exposes it). I must emphasize that what follows is a quick, mostly practically oriented, overview of data structures, a core topic of any computer science curriculum and the object of full courses in its own right in any university (CS203 at SUSTech).



DATA

When you think about it, "information" is nothing more than "data" to which some meaning is attached. This really tells how much data is at the heart of systems and why the old "EDP" name may not be as glamorous and cool as "IT" but is still in fact what we are doing.

meaning

Information

As data is so important, it's vital to find efficient ways to handle it.

Where is the information we have lost in data?

Where is the wisdom we have lost in knowledge?
Where is the knowledge we have lost in information?

T.S. Eliot
(1888-1965)

The American poet TS Eliot famously lamented the loss of wisdom and knowledge. All the work going on about "big data" is mostly about not losing information in a sea of data.

DATA DATA
DATA
DATA

In fact, information comes not from one single, isolated piece of data, but from a combination of pieces of data.

meaning

If I tell you "C", it's data, not information. "Dennis Ritchie" is also data. But "Dennis Ritchie created C" is information (I could add "in 1970").

Information

This is why data structures that link pieces of data together are so important.

SMART ways to store and handle data

Simple variables

Arrays one dimension, two dimensions or more

struct

Arrays of **struct**

What we have seen so far as ways to handle and store data in a program is in fact rather limited.

Arrays are nice but ...

... you need to predict how many slots you will ever need.

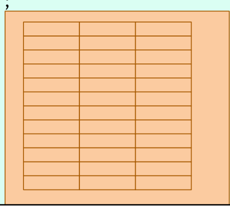
Arrays, typically, require a predefined size (which can be set from a variable). Very often, you oversize them to avoid problems. Memory is far less an issue than in the days of Dennis Ritchie, but as the volumes of data to handle also have exploded since then, it remains an issue.

Manage memory **dynamically**

We have seen that we can manage data dynamically, which is a great improvement.

Combining statically and dynamically allocated memory

```
typedef struct place {
                      char   *placename;  8 bytes
                      double latitude;    8 bytes
                      double longitude;   8 bytes
                     } PLACE_T;
```



```
PLACE_T places[MAX_PLACES];
```

I'd like to point out that you can mix static and dynamic memory.

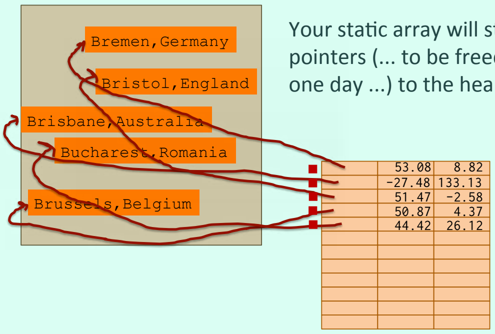
```
...
"Bremen,Germany",53.08,8.82
"Brisbane,Australia",-27.48,153.13
"Bristol,England",51.47,-2.58
"Brussels,Belgium",50.87,4.37
"Bucharest,Romania",44.42,26.12
"Budapest,Hungary",47.5,19.08
"Buenos Aires,Argentina",-34.58,-58.37
"Buffalo,NY,USA",42.92,-78.83
"Cairo,Egypt",30.03,31.35
"Calcutta,India",22.57,88.4
"Calgary,AB,Canada",51.02,-114.02
"Canton,China",23.12,113.25
"Cape Town,South Africa",-33.92,18.37
"Caracas,Venezuela",10.47,-67.03
...
```

If you read data from a file such as this one, you can oversize your array (because each element is rather small) and dynamically allocate the names that take space.

```
int i = 0;
char line[MAXLINE_LEN];

...
while (fgets(line, MAXLINE_LEN, fp)) {

     ... Scan line ...

     places[i].latitude = ...;
     places[i].longitude = ...;
     places[i].placename = strdup(...);
     i++;
 }
```

It's very easily done with `strdup()` (You NEED a copy. Think why if it's not obvious to you)

Combining statically and dynamically allocated memory



Your static array will store pointers (... to be freed one day ...) to the heap.

BIG ADVANTAGE: SORTING

```
#define MAX_CITY_LEN     35

typedef struct place {
                      char   placename[MAX_CITY_LEN];
                      double latitude;
                      double longitude;
                     } PLACE_T;
```

35 + 16 = 51 bytes
alignment -> 52

This way of storing data is quite interesting for sorting (and you don't need to wonder how long a name can be). If you exchange elements that contain names, you move units of 52 bytes each time.

BIG ADVANTAGE: SORTING

```
typedef struct place {
                      char  *placename;
                      double latitude;
                      double longitude;
                     } PLACE_T;
```

24 bytes only

If your elements contain pointers, you only need to exchange pointers, not what they point to. Exchanging elements will mean shifting units of 24 bytes only (on a 64bit machine). Shuffling fewer bytes means that it can be done much faster.

So far, two tactics:

Give a fixed size to an array, and store data, possibly allocating some memory dynamically

Read size required, allocate array, and store data

What if we don't know the array size in advance?

If we have the array size wrong and too small, we are toast if it wasn't created completely dynamically. However, if it was created dynamically, we have a way out with a function I have just quickly mentioned so far, which is `realloc()`. This function allows you to resize an area in memory.

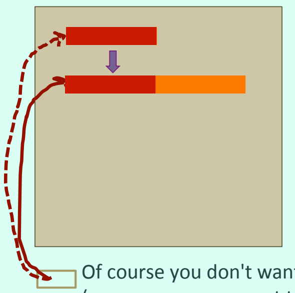
```
#define ELEMENT_SET     100

static PLACE_T *G_places = NULL;
static int      G_places_elements = 0;

if ((G_places = (PLACE_T *)malloc(sizeof(PLACE_T)
                                  * ELEMENT_SET))
                  == NULL) {
   // Quit
   ...
}
G_places_elements = ELEMENT_SET;
```

Initially, you give your array a size that looks OK in most cases. You record this size in a variable. When you add elements to your array, the index keeps a count of how many "slots" are used (you'll note that I'm using global variables that are local to a file).

```
if ((G_places = (PLACE_T *)realloc(G_places,
                                   sizeof(PLACE_T)
                                  * (G_places_elements
                                     + ELEMENT_SET)))
                  == NULL) {
   // Quit
   ...
}
G_places_elements += ELEMENT_SET;
```

When every single slot is used, you call `realloc()`, pass to it the address of the memory area, and ask for a bigger size. It will return to you the new address. One important point is that you'll need to free this new address, but no longer need to free the previous one, because `realloc()` frees it.

What `realloc()` does it that it looks like `malloc()` for a free, big enough memory area in the heap that can be assigned to you and reserves it. But it doesn't stop here.



It then uses the original pointer to copy what was at the old place to the new place, then it frees the old place before returning to your program the address of the new one.

Of course you don't want to do that too often (memory management takes time), but your program won't have to stop when data is more than expected.

When we store data, we want to find it.

Data is easier to find when ordered.

Now when we store data, it's you usually to retrieve it later, and retrieval should be efficient. You cannot efficiently retrieve something that isn't ordered, in a way or another (think of what an unordered dictionary or directory would be). Rules for sorting Chinese characters may be different from rules for sorting Latin characters, there are still rules. Arrays aren't so good for keeping order. You can sort them, of course, but as sorting is a costly operation this is something you want to do only once. If you add items to an array in a random order, it becomes very inefficient.

if the array cannot accommodate one more element
 reallocate something bigger
find the place in the array where the element must be inserted
if it's not the last place
 for all elements from the last one down to the following one
 copy element from position n to position n + 1
insert the new element

Ouch.

If you want to insert an item "at the right place" in an array, you basically have to move everything that follows to make room. It hurts.

Data Structures

This is why people have thought that instead of connecting two successive elements implicitly by the closeness of their memory location as in arrays, two successive elements might be better explicitly connected by pointers. Welcome to the wonderful and magical world of Data Structures.

Data structures in C are a Do-It-Yourself world. You define a struct, and connect elements between them or walk collections through pointers.

Data Structures

```
typedef struct suitable_struct
        {
          struct suitable *next_call;
        } SUITABLE_STRUCT_T;

SUITABLE_STRUCT_T *my_struct;
```

The most interesting data structures are usually structures that contain at least one pointer to another node.

The "collection" is a pointer to its first element.

Sometimes two pointers ...

```
typedef struct place {
        char   *place_name;
        double latitude;
        double longitude;
} PLACE_T;
```

One element

Node

When you create an array, you reserve a big lump of memory in which the various elements are stored one after the other. The idea here is to consider each element individually (usually you talk of "node")

Solution: pointers

name becomes mandatory

```
typedef struct place {
            char   *place_name;
            double  latitude;
            double  longitude;
            struct  place *next;
         } PLACE_T;
```

Beware that as the compiler reads lines in order, when it comes across the pointer it knows what a `struct place` is but not what a PLACE_T is

Not usable in this structure

```
typedef struct place_list {
            PLACE_T   *first_element;
            int        place_count;
         } PLACE_LIST_T;
```

Here it works