CS205

C / C++

Stéphane Faroult
faroult@sustc.edu.cn

Wang Wei (Vivian) vivian2017@aliyun.com

Writing/reading any type of file

```
c = fgetc(fp);

fputc(c, fp);
```

With both text and binary files, C allows to process byte-by-byte, reading one character at a time (you may think it is slower, but in fact at the lower level many bytes are read at once and are just returned one by one, so in fact it's quite efficient). You may want to do this when you aren't interested in the content of files, to make a copy of a file for instance.

Writing to / reading from a binary file

Dumping memory to a file

```
fwrite(ptr, unitary_size, num_elem, fp);

fread(ptr, unitary_size, num_elem, fp);
```

Beware with fread of not overflowing the memory buffer

return number of elements written/read

Binary files basically store copies of bytes in memory. You pass a pointer, assumed to be the address of an array, the size of each element and the number of elements.

Writing to / reading from a binary file

Dumping memory to a file

Problem **ERROR** or **end of file?**

```
fread(ptr, unitary_size, num_elem, fp);

feof(fp);
ferror(fp);
```

Those functions allow you to know whether you got a NULL (or EOF) because there was nothing more to read, or because of an error.

Historical binary files

Binary files used to be very important to store massive amounts of data. Today databases (which manage files) are used for this purpose. Most binary files that you will encounter will probably be media files, not large collections of data.

When all records are same-sized structures, getting to one in particular is easy with function fseek()

→ struct

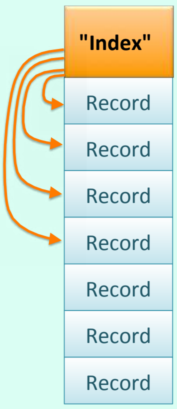
Easy to update

"direct access"

```
fseek(fp,
      offset,
      origin)
```

from beginning SEEK_SET
from current SEEK_CUR
from end SEEK_END

```
fseek(fp,
    (n – 1) * sizeof(record),
    SEEK_SET)
```



When records may differ in size, or to search say by name an "index" may store the offsets to each record.

Database products often use this to locate collections of data stored in "tables" actually composed of blocks (records)

Other File Operations

You can delete a file in a program	unlink()
You can check multiple accesses	flock()

Directory Operations

`#include <dirent.h>`

You can open a directory, `opendir()`

read directory entries (struct), `readdir()`

close the directory. `closedir()`

Side Note

Some libraries provide special structures that replace FILE* and some functions that replace the standard ones for reading special types of files.

Examples: reading ZIP or XML files

`presentation.pptx`

Many file formats are actually zipped text files. This is the case for Microsoft Office products, among others. The Unix command od with the –c (character) flag allows you to dump a binary file.

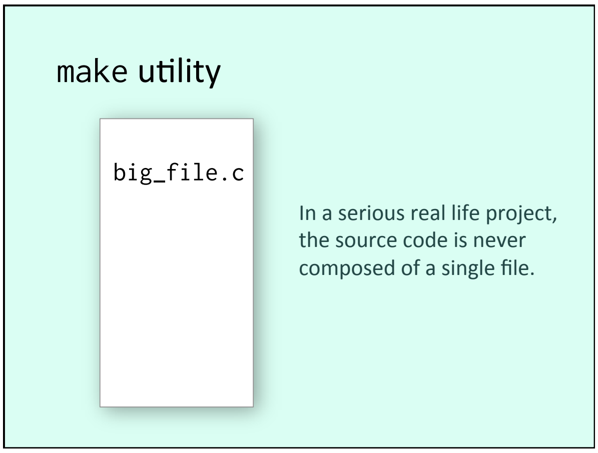
```
od –c presentation.pptx | head
```

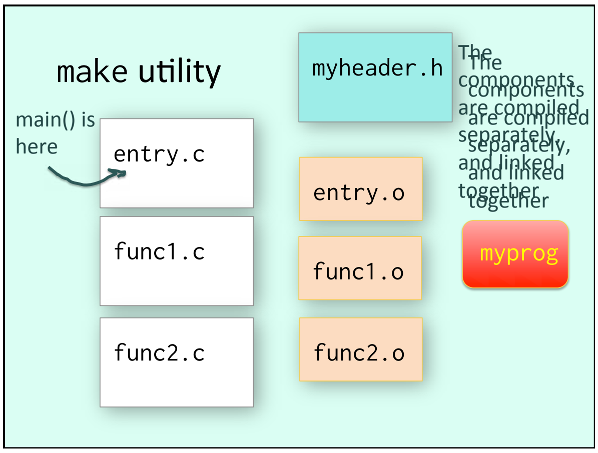
```
0000000   P   K 003 004 024  \0 006  \0  \b  \0  \0  \0   !  \0   L 242
0000020 242 335 033 002  \0  \0   - 020  \0  \0 023  \0  \b 002   [   C
0000040   o   n   t   e   n   t   _   T   y   p   e   s   ]   .   x   m
0000060   l     242 004 002   ( 240  \0 002  \0  \0  \0  \0  \0  \0  \0
0000100  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0
*
0001060  \0  \0  \0  \0  \0  \0  \0  \0  \0      ** 315   n 333   0  \f
0001100 200 357 003 372 016 206 256   E 254 244 353 272   n 210 323   ö
0001120  ** 236 366   S 240 335 003 250   6 355   h 263   %   A   b 262
0001140 346 355   G 333   i   * 030   ^ 275 304   6 222   K  \f   I   &
```

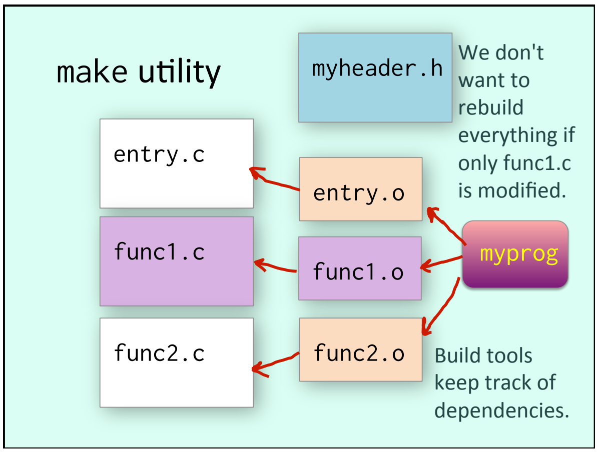
When od –c shows to you PK as the first two bytes in a file, it means that it really is a Zip archive (Zip was created by a programmer named Phil Katz)

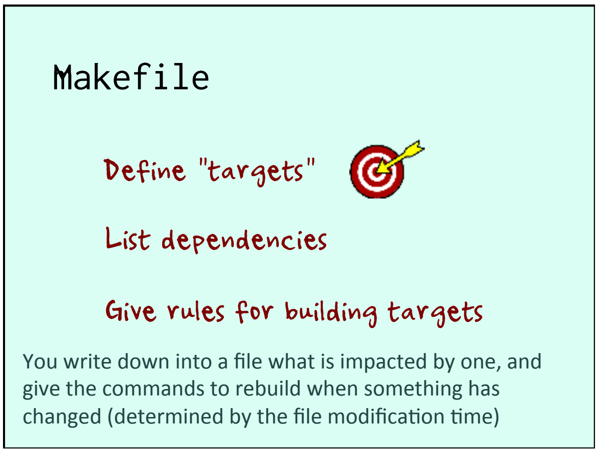
`unzip presentation.pptx`

You can safely unzip it and see what it contains (many subdirectories and files). Same story with spreadsheets or Word files, and many other companies apply the same technique. With the right libraries, you can read and process these files. This is how I generate the index for the course notes: the indexed words are tags that I add to the "presenter notes" in the commented slides ... A C program reads them, get the slide number, and derives the page number from the number of slides per page in the handout.


make utility
big_file.c
In a serious real life project, the source code is never composed of a single file.


make utility
myheader.h
The components are compiled separately, and linked together
main() is here
entry.c
func1.c
func2.c
entry.o
func1.o
func2.o
myprog


make utility
myheader.h
We don't want to rebuild everything if only func1.c is modified.
entry.c
func1.c
func2.c
entry.o
func1.o
func2.o
myprog
Build tools keep track of dependencies.


Makefile
Define "targets"
List dependencies
Give rules for building targets
You write down into a file what is impacted by one, and give the commands to rebuild when something has changed (determined by the file modification time)

makefile

```
LIBS = -lm

all: myprog

myprog: entry.o func1.o func2.o
	gcc -o myprog entry.o func1.o func2.o $(LIBS)
```

This is what you must have before you can build the target.

Implicit rules

tab

This is a very simple example. There are implicit rules - if "make" needs blah.o and finds blah.c in the current directory, it knows how to do it. Targets are separated by empty lines ("all" is the default target). $(name) is subsituted in the rule. Commands to build a target must start with a tab.

makefile

NO spaces

```
LIBS = -lm

all: myprog

myprog: entry.o func1.o func2.o func3.o func4.o \
        func5.o func6.o
    gcc -o myprog entry.o func1.o func2.o func3.o \
        func4.o func5.o func6.o $(LIBS)
```

When you have long lines, you must terminate them with a backslash (\) followed by NOTHING, not even a space. It indicates that the next line is the continuation of the current one.

```
$ make
cc    -c -o entry.o entry.c
cc    -c -o func1.o func1.c
cc    -c -o func2.o func2.c
gcc -o myprog entry.o func1.o func2.o –lm
$
```

I have highlighted here the default rules that have been applied to build my program when I typed 'make'; they weren't written in the makefile (personally, I tend to write everything, partly as documentation). You'll also notice here (run on my Mac) that the compiler called by default was 'cc', not 'gcc'. Not a problem, and it could be changed.

makefile

```
CFLAGS = -Wall
LIBS = -lm

all: myprog

myprog: entry.o func1.o func2.o
    gcc -o myprog entry.o func1.o func2.o $(LIBS)
```

You can also define a CFLAGS variable that applies special options. I warmly recommend -Wall that stands for "Warning all" and detects many potentially dangerous mistakes. Another commonly used flag is -I to tell where to look for (your) header files if they are in a subdirectory somewhere.

makefile

```
CFLAGS = -Wall
LIBS = -lm

all: myprog

myprog: entry.o func1.o func2.o
   gcc -o myprog entry.o func1.o func2.o $(LIBS)

%.o: %.c
   gcc $(CFLAGS) -c $< -o $@

clean:
   -rm myprog
   -rm *.o
```

You can also define "template rules", in which $@ stands for the target and $< for the dependency, which are a way to fine-tune default rules.

If a rule starts with a minus sign, it doesn't interrupt the build if it fails (for instance here if there are no .o files to remove)

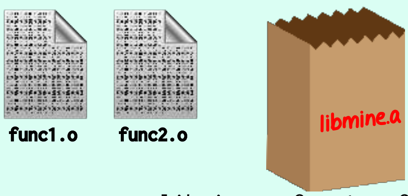
To build a specific target, you pass its name on the command line. Otherwise what is built is "all".

```
$ make clean
rm myprog
rm *.o
$ make
gcc -Wall -c entry.c -o entry.o
gcc -Wall -c func1.c -o func1.o
gcc -Wall -c func2.c -o func2.o
gcc -o myprog entry.o func1.o func2.o −lm
$
```

Template rules in action.

Libraries

In very large projects, .o files are often packed into libraries. You link with the library instead of the individual .o files.




```
ar -rs libmine.a func1.o func2.o
```

-r means replace if present, -s means index (to speed up linking). A .a unix library is static, it can also be created dynamic (shareable between processes running at the same time).

---

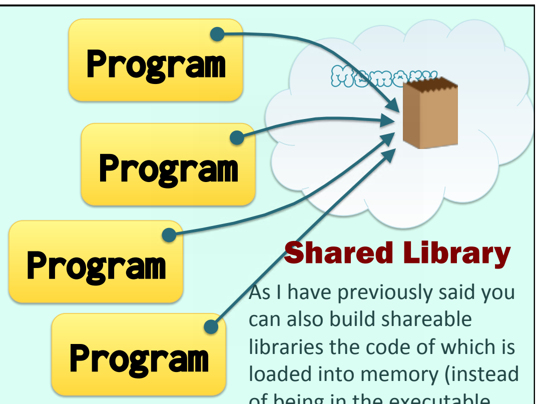
makefile

```
CFLAGS = -Wall
LIBS = -lm

all: myprog

myprog: entry.o libmine.a
   gcc -L. -o myprog entry.o -lmine $(LIBS)

%.o: %.c
   gcc $(CFLAGS) -c $< -o $@

libmine.a: func1.o func2.o
   ar -rs libmine.a func1.o func2.o

clean:
   -rm myprog
   -rm *.o
   -rm *.a
```

-lxxxx = link with libxxxx

Template rules in action.

Here is the modified makefile. The additional flag -L is for the linker only, and says to also look into the specified directory for libraries.



Shared Library

As I have previously said you can also build shareable libraries the code of which is loaded into memory (instead of being in the executable program)

Keeps no state

Reentrant

.so

.dll

.dylib

OS X

Shareable libraries must have some characteristics, mainly of not storing states, which is called being "reentrant". Extensions of dynamic libraries aren't the same ones on all system. On Windows, they have long been a standard feature.

`static` and `extern` functions

`static` = class function

~~class~~ file

`static` ≈ private

`extern` ≈ public

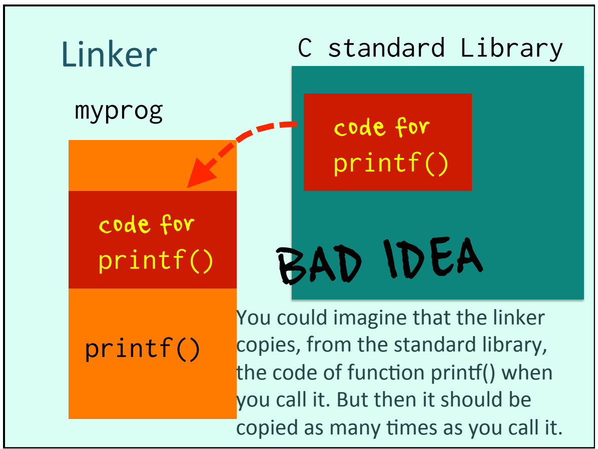
Functions can be extern or static. Although "static" also exists in Java, the meaning is C is different and only remotely related.

`static` and `extern` functions

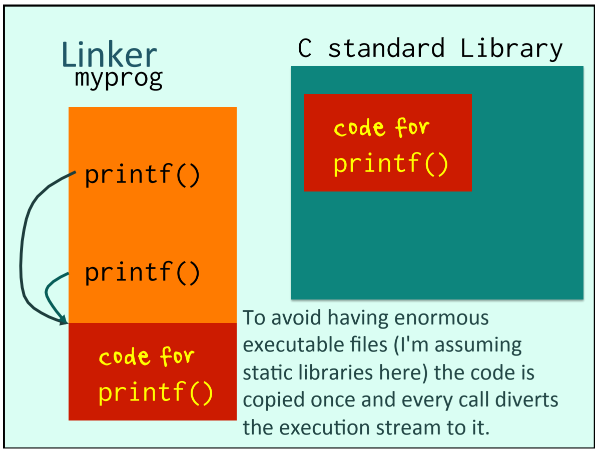
When a method is `static` in Java, it means that it's a class method; you don't need to instantiate an object of the class to call it. Typical examples are `main()` and the `Math` methods or methods in classes such as `Integer` that are just wrappers around plain basic types.
`static` functions in C can only be called by functions in the same `.c` file. They are invisible to the linker, that won't find them. Functions that are declared as `extern` (the default) ARE findable by the linker. You have in C some kind of primitive encapsulation, where functions are "public" or "private" with relation to a source `.c` file instead of an object.

What happens when you call a function?

We have seen that the linker finds the code of functions that (for many of them) to add it to your program. Looking at how it works and what happens when you call a function is very instructive for understanding how your program runs and why you should so this and not that if you don't want to see it crash in the middle of a run.



You could imagine that the linker copies, from the standard library, the code of function printf() when you call it. But then it should be copied as many times as you call it.



To avoid having enormous executable files (I'm assuming static libraries here) the code is copied once and every call diverts the execution stream to it.

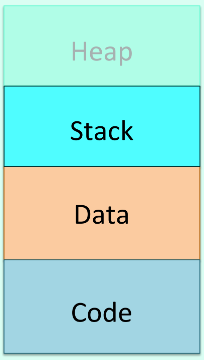
```c
int gcd(int n1, int n2) {
  // Greatest common divider, Euclid
  int r;

  r = n1 % n2;
  while (r) {
     n1 = n2;
     n2 = r;
     r = n1 % n2;
  }
  return n2;
}
```

Let's take a simple example with this function, which computes the greatest common divider (GCD) ov two integer values.

```
int a;
int b;
int c;

...
a = 12;
b = 3;
c = gcd(a, b);
printf("Value of c : %d\n", c);
...
```

must jump to the instructions in the function
must transmit values
must come back!

There are several problems to solve here. After executing the assignment of value 3 to variable b, the program will continue in sequence but the next instruction isn't as plain an instruction as an assignment: it's a function call. When it builds the program, the linker will insert here an instructions that says "now run instructions that start at that place", with a reference to where it (the linker) has put the code of function `gcd()`. This is not all: the functions expect two parameters, how are we going to transmit them? Moreover, when the code of the function is finished, how to tell it where to return the value, and that we want to call `printf()` next? We could call the function from many places in our program, and it might need to return and execute completely different instructions.



von Neumann Architecture

This is where the stack in the von Neumann architecture comes into play. The stack is a region in memory that is used in a dynamic way, to exchange information with functions that are called. The information is put by the caller in the stack, where functions know to find them. They will put back return values there too.

```
int a;
int b;
int c;
...
a = 12;
b = 3;
c = gcd(a, b);
printf("Value of c : %d\n", c);
...
```

This is supposed to be the stack

12
3
<return address>

✓ **must come back!**
✓ must transmit values

Before jumping to the function, the linker will add instructions to store on the stack where to go when the function returns, and the values that are passed to it.

```
int gcd(int n1, int n2) {
  // Greatest common divider, Euclid
  int r;
  r
  while (r) {
     n1 = n2;
     n2 = r;
     r = n1 % n2;
  }
  return n2;
}
```

12
3
<return address>

When the function is called, it knows that it finds all the information it requires on the stack (and as the parameter values are passed, it doesn't matter whether they are called one name here and another name there; what matters is knowing their type and in which order they have been put on the stack)

```
int gcd(int n1, int n2) {
  // Greatest common divider, Euclid
  int

  r = n1 % n2;
  while (r) {
     n1 = n2;
     n2 = r;
     r = n1 % n2;
  }
  return n2;
}
```

In fact, local variables are ALSO created on the stack. When we return, they are gone.

12
3
<return address>

Why is it called "stack"?

It's because you very often call functions from within functions; and so the process is repeated a number of times, adding new return addresses, parameter values and local variables each time like adding plates to a stack. When you return from a function, it's like removing a plate and you return to the calling function with its local variables and so forth;

Flickr: Jules

[variables in f3]
[parameters for f3]
<return address to f2>
[variables in f2]
[parameters for f2]
<return address to f1>
[variables in f1]
[parameters for f1]
<return address to main>

```
int f2(...) {
  ...
  a = f3(...)
  ...
  return x;
}
float f1(...) {
  ...
  n = f2(...)
  ...
  return val;
}
int main() {
  ...
  x = f1(...)
  ...
  return 0;
}
```

Here is what the stack looks like when main() calls f1() that calls f2() that calls f3()

Parameters modified in a function don't affect the caller.

Only the returned value is known by the caller.

Because functions find on the stack copies of the values that are passed to them, they can modify them as much as they want, when you return to the caller you retrieve the state of the caller; the only effect of the function for the caller is what the function returns (however, things are somewhat different when what you pass to a function is a pointer. If the pointer is modified there will be no change for the caller, but if what is at this address is modified it will be different)

To see what happens with pointers, let's say that we want to write a function that takes a string of Latin characters and turns every letter at the beginning of a word into uppercase and every other letter into lowercase. This is a process frequently applied in English to book titles, movie titles, and chapter headings in books (only to important words), for instance "Three Kingdoms".

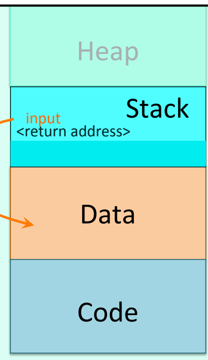
```
char *initcap(char *input) {
  char output[MAX_LEN];
  int  i = 0;
  char not_after_letter = 1;

  if (input != NULL) {
     while (input[i] != '\0') {
        ...
        i++;
     }
     output[i] = '\0';
     return output;
  }
  return NULL;
}
```

FAIL

If you declare an array in your function to store the modified string and return this array, your program will fail. Here is why.

Address

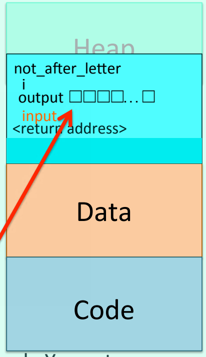
```
char *initcap(char *input) {
  char output[MAX_LEN];
  int  i = 0;
  char not_after_letter = 1;

  if (input != NULL) {
     while (input[i] != '\0') {
        ...
        i++;
     }
     output[i] = '\0';
     return output;
  }
  return NULL;
}
```



When you call the function, you put in the stack return address plus a pointer to the string to change, stored elsewhere

```c
char *initcap(char *input) {
  char output[MAX_LEN];
  int  i = 0;
  char not_after_letter = 1;

  if (input != NULL) {
     while (input[i] != '\0') {
        ...
        i++;
     }
     output[i] = '\0';
     return output;
  }
  return NULL;
}
```

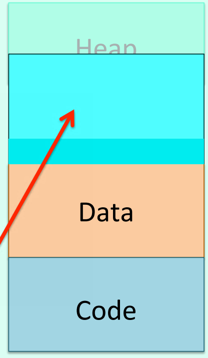
Address



Local variables are stored on the stack. You return an address that points to the stack. But when you return, the stack is freed.

So the caller gets back the address of something that may be affected to something else - an invalid address.

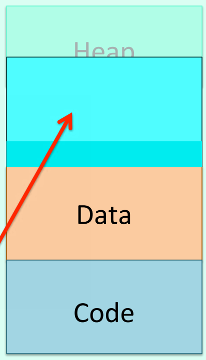
This is what the caller gets.



Address of vanished data

... the caller gets the address of something that has the same life-span as the function, which is no longer.

This is what the caller gets.



Address of vanished data
Invalid address

You CAN return a single value (`char`, `int`, `long`, `float`, `double` ...) or a (small) `struct`.

You CANNOT return an array declared into the function – if you return an address, it must point to something that is valid for the caller.

There is no problem returning a local int for instance, because its value is copied back to the stack. With a local array, what is copied back is just the address of the first element. You can return an address, but to something that outlives the function.

There are **4** solutions.

There are four different solutions for return a string (or any array) from a function in C. Some are better than others.

1 Transmit pointers

If you have allocated memory previously to calling the function, and say to the function not only "read from this address" but "store the modified string at this address", you won't have any problem.

```c
void initcap(char *input, char *output) {
  int  i = 0;
  char not_after_letter = 1;

  if (input != NULL) {
     while (input[i] != '\0') {
        ...
        i++;
     }
     output[i] = '\0';
  }
}
caller() {
  char ugly_name[MAX_LEN];
  char pretty_name[MAX_LEN];
  ...
  initcap(ugly_name, pretty_name);
  ...
}
```

nothing returned

address address

Both "input" and "output" point to memory reserved by the caller (possibly in the stack) that will survive when returning from the function.

Must, as always, be careful with pointers ...

Besides, this way of proceeding looks unnatural to many people who like functions to "return something", instead of modifying memory in a way that looks incomfortably like side-effects.

You can also pass the address of a **short**, **int**, **float** ...

dereferenced in the function

If you want to modify something else than a string, you can apply the same principle to any type of data. The function always gets a copy on the stack. By passing a copy of the address, the function becomes able to modify the real place in memory.

Remember?

This is why you pass an address to scanf, to enable it to write at the address where the integer is stored.

```
printf("Enter an integer : ");
scanf("%d", &my_int);
```

```
char *fgets(char *str, int size, stdin)
```

No such problem with fgets(), that gets a string, therefore the name of a char array, therefore an address where to store what it reads ...

If a function expects a pointer to a numerical value, you can't call it with a constant argument.

```
int my_func(int a, int *b_ptr) {
   ...
}
```

Yes NO!

```
n = my_func(3, 5);
```

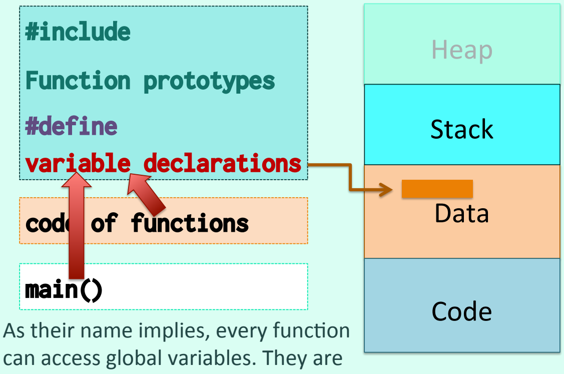
If a function expects a pointer to a numerical value, you can't call it with a constant argument.

```
int my_func(int a, int *b_ptr) {
   ...
}
some_int = 5;
n = my_func(3, &some_int);
```

This will work.

2 Global variables

Global variables (such as `errno`) are another way to work around the problem of memory being freed up when returning from a function. Global variables are reserved for the whole life-span of the program.



As their name implies, every function can access global variables. They are declared outside functions (variables declared in `main()` are not accessible from other functions)

```
char G_output[MAX_LEN];

char *initcap(char *input) {
  int  i = 0;
  char not_after_letter = 1;

  if (input != NULL) {
     while (input[i] != '\0') {
        ...
        i++;
     }
     G_output[i] = '\0';
     return G_output;
  }
  return NULL;
}
```

I usually prefix the name of global variables with G_ (personal habit).

Drawbacks:

The general agreement is that global variables are evil.

- Side-effects of other functions
- Can be "hidden" by a similarly named local variable

  if you don't give them special names.

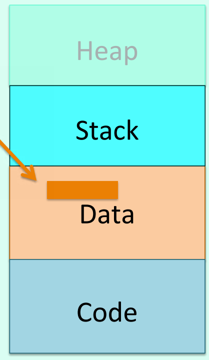
- Multi-threading (advanced!)

Note that global variables the declaration of which is prefixed by "static" are only global to the functions in the FILE, which is a far lesser sin.

3 Static variables

In fact "static" in a C program has two different, but not completely unrelated, meanings. For variables, it means that they are stored in a memory area that is reserved by the compiler and will be there as long as the program is running. For variables and functions it also means "private" to the file.

```c
char *initcap(char *input) {
  static char output[MAX_LEN];
  int  i = 0;
  char not_after_letter = 1;

  if (input != NULL) {
     while (input[i] != '\0') {
        ...
        i++;
     }
     output[i] = '\0';
     return output;
  }
  return NULL;
}
```



Same storage as with a global variable, but invisible outside the function.

Drawbacks:

- Too many may waste a lot of memory
- Multi-threading (advanced!)

Note that some standard C functions uses something of that kind. Think of one function that you have seen, `strtok()` that can be used to parse a string. At the first call it stores the string to tokenize. For the next calls you should pass NULL instead of the string, and the function returns successive tokens.

4 Dynamic memory

Memory allocation is mostly hidden from view in languages such as Java - it still takes place, of course, but you have no control on it. All you know is new, which is called "object instanciation" (spoiler: it reserves memory) C micro-manages it, so you really do whatever you want. On the other hand, if you do it badly there is no safety net, which forces you to understand what is going on.

C allows to ask the system to **reserve memory** where to store our data.

That's how it's called.

Memory allocation

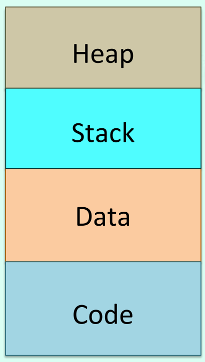
new

Java does it too, but doesn't advertise it for what it really is.

There is one very serious difference with Java though, which is that Java has a built-in garbage collector that cleans up the mess after you. In C and C++ you must do it yourself.

Memory allocation

BUT there is **no garbage collection** in C (or C++)



von Neumann Architecture

Memory allocation takes place in the last region of the von Neumann architecture, the heap. In reality, stack and heap are in reverse order with regard to memory addresses, but I'm interested in principles, not implementation details, and I find it easier to show a stack that grows upwards rather than downwards.

Four main functions to know

```
#include <stdlib.h>

void * malloc(size_t size);
     void * realloc(void *ptr, size_t size);

void free(void *ptr);
```

There is also calloc() that initializes at the same time.

```
#include <string.h>

char * strdup(char *str);
```

strdup() duplicates, and combines memory allocation with copy.

void * ?

You know void from Java (methods that return nothing return void). But what is a void pointer?

Flickr: San Drino

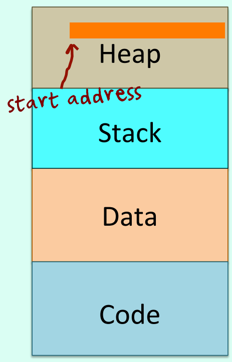
void * = address of "something"

type unknown

size unknown

Must cast !

If you need to code something really, really dirty, that's the tool for you.



malloc(1000)

returns start address or NULL

malloc() finds free room for you in the heap, and returns a pointer to the beginning of the memory chunk, of NULL if it fails to find what you are asking for.

```
char *initcap(char *input) {
  int   i = 0;
  char  not_after_letter = 1;
  char *output = NULL;

  if ((input != NULL)
      && ((output = (char *)malloc(strlen(input) + 1))
                 != NULL)) {
     while (input[i] != '\0') {
        ...
        i++;
     }
     output[i] = '\0';
     return output;
  }
  return NULL;
}
```

And here you are returning an address pointing to something outside the stack. In an object-oriented language, instantiating an object actually performs a `malloc()`, which is why they allow you to return objects from functions. An object is in fact a pointer to a structure. But hush, don't repeat it.

What is ALLOCATED must be FREED when you no longer need it.

Once again, there is in C no garbage collector doing the cleaning for you. You must take care of housekeeping chores yourself.

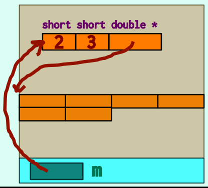
```
#define MAX_ELEMENTS    2500  // enough for 50x50

typedef struct matrix {
           short rows;
           short cols;
           double cells[MAX_ELEMENTS];
        } MATRIX_T;
```

Managing memory dynamically allows for great flexibility. You are no longer constrained by predefined constants.

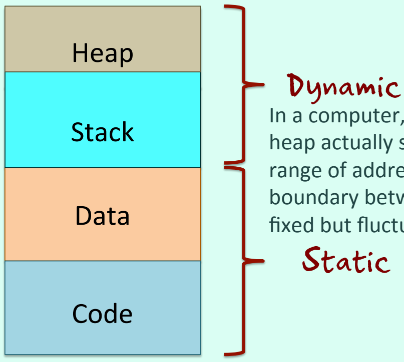
```
typedef struct matrix {
           short rows;
           short cols;
           double *cells;
        } MATRIX_T;
```

The number of cells that you need will be allocated on the fly, which will both reduce memory waste and allow to cope when there is more data than usual.

```c
MATRIX_T *new_matrix(short r, short c) {
  MATRIX_T *m = NULL;

  // WARNING: no error checking !
  m = (MATRIX_T *)malloc(sizeof(MATRIX_T));
  m->rows = r;
  m->cols = c;
  m->cells = (double *)malloc(sizeof(double) * r * c);
  return m;
}
```



You first allocate a structure, then within the structure you allocate an array big enough to hold the values.



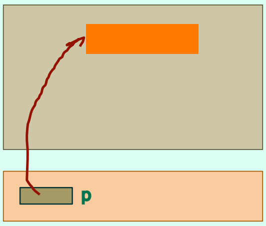
In a computer, the stack and the heap actually share the same range of addresses and the boundary between the two isn't fixed but fluctuates.

Warning: not to scale

Memory is normally freed when you exit a program, **BUT**

```c
void free(void *ptr);
```

```c
char *p = NULL;
p = (char *)malloc(len);


free(p);
p = NULL;
```



free() takes a pointer to a memory chunk in the heap and releases it. Note that it doesn't reset the pointer itself; you should do it.

You can only free memory that you have allocated.

Otherwise, ugly program crash.

mem.c

```c
#include <stdio.h>
#include <stdlib.h>

int main() {
    int   val = 42;
    int  *ptr = &val;

    free(ptr);
    return 0;
}
$ gcc mem.c -o mem
$ ./mem
mem(3913,0x7fff7b906310) malloc: *** error for object
0x7fff58106be8: pointer being freed was not allocated
*** set a breakpoint in malloc_error_break to debug
Abort trap: 6
$
```

Once freed, memory is no longer yours.

Otherwise, ugly program crash. Or perhaps that you'll get rubbish.

mem2.c

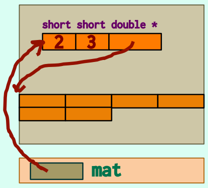
```c
#include <stdio.h>
#include <stdlib.h>

int main() {
    int  *ptr;

    ptr = (int *)malloc(sizeof(int));
    *ptr = 42;
    printf("I have put %d in the heap\n", *ptr);
    free(ptr);
    *ptr = 15;
    printf("Now I have %d\n", *ptr);
    return 0;
}
```

ptr still contains an address but it's no longer reserved and anything can happen.

When you free memory, you mustn't leave cleaning half done.

Think of others. Release EVERYTHING that you no longer need.

```c
MATRIX_T *mat = NULL;
mat = new_matrix(2, 3);
```



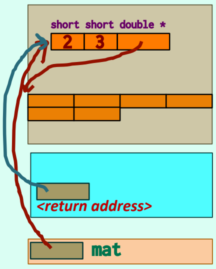
... doing stuff ...
... until I no longer need **mat**.

```c
free(mat);
```

WRONG!

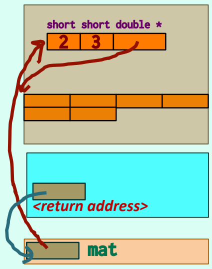
If you only free the structure, the memory that you have reserved for the cells will be still there and reserved to you, and you will be no longer able to access it. Lost for everybody.

```c
void free_mat(MATRIX_T *m) {
   if (m) {
      if (m->cells) {
         free(m->cells);
      }
      free(m);
   }
}
```



That's how you should do it.

```c
void free_mat(MATRIX_T **mp) {
   if (mp && *mp) {
      if ((*mp)->cells) {
         free((*mp)->cells);
      }
      free(*mp);
      *mp = NULL;
   }
}
```



By passing a pointer to a pointer, you can even reset the pointer in the function.

Any option is OK as long as you know what you are doing.

Use the first method if pointers to pointers give you headaches. It requires some practice to get accustomed (we'll see them again).