0x0d2C

------
May your signals all trap
    May your references be bounded
All memory aligned
    Floats to ints round

remember ...

Non -zero is true
    ++ adds one
Arrays start with zero
    and, NULL is for none

For octal, use zero
    Ox means hex
= will set
    ==means test

use -> for a pointer
    a dot if its not
? : is confusing
    use them a lot

a.out is your program
    there's no U in foobar
and, char (*(*x())[][])() is a function returning a pointer to an array of pointers to functions returning char.
There is an easy trick for reading complex C types.

- Start by peeling back one layer at a time.
- Use the meaning of *, and the fact that * and & cancel.
- Use the meaning of (*x)(...) and the fact that x(...) becomes the outer type.

Example:

```c
int ***c;
```

means

```c
***c is an int
```

which means

```c
**c is of type int *
```

which means

```c
*c is of type int **
```

which means

```c
c is of type int ***
```

(in each step, we apply & to both sides)

Example:

```c
int ** (*foo) (int x, int y);
```

foo(5,7) is of type int **

*foo(5,7) is of type int *

**foo(5,7) is of type int
Header files
  Declarations, not definitions.
  The C preprocessor and \#defines

C types
  Typedefs
  Portable primitive types
  Pointers to function
  Storage classes: extern, static.
Header files

So far, we've concentrated on reading manual pages. Today, we consider the larger issue of reading C header files (.h)

Two issues:

What can we write?
Why do we write it?
Why have header files?

Declare things that exist elsewhere (e.g., in the C library or system calls) that you can use.
Declare types needed in order to interact with them.
Allow multiple programs to #include the same header to use the same external functions or types.
Example: stdout.h: declares printf, fopen, fclose, ...
There are two kinds of C source files

Header files: **declare** things to be used by others (.h).

Source files: **define** things that take up memory (.c).

Reason for the split: one should only **define something once**, but one can **declare it in every file that uses it**.

**Definitions consume memory.**

**Declarations do not.**
Header files

- end in .h
- mostly kept in /usr/include
- describe the types of arguments needed for subroutine calls.
- Some apparent functions are macros, substituted inline wherever calls appear.
  (Reason: speed! Function calls take time!)

What's in a header:

A header shouldn't define anything.
(in other words, it should not create functions or data that use storage)

A header should declare everything.
(in other words, it describes the structure of already created things)
Exception: macros aren't defined until used, so they can be declared in headers.

General contents of header file:

#defines: macros
structure and union declarations
enumerated type declarations
declarations of (global) variables and functions
typedefs (compiler macros)
inline functions (compiler macros)

A subtle distinction:
inline int foo() { ... }

should be in a header file, while

int foo() { ... }

should not be there.

Reason: an inline function is a macro in disguise. It is actually a declaration.

Another subtle distinction

extern int foo; /* declaration */

versus

int foo; /* definition */
A typical source file foo.c
    #include <stdio.h>
    void foo(const char *s)
    { printf("hello %s\n",s); }

A header file foo.h that declares that:
    void foo(const char *);
    or
    extern void foo(const char *);

Any program that wishes to use foo:
    #include "foo.h"
    ...
    foo("harry");

Watch out:
    If a subroutine is undeclared at time of use,
    **undeclared types default to int!**

Caveats: <> means search include path: /usr/include....
    "" means search current directory or absolute path.
A header file contains:

Preprocessor directives:

```
#define TRUE 1
#include "foo.h"
#ifdef TRUE
...
#endif /* TRUE */
```

declarations:

```
extern int k;
extern void foo(int);
static double d;
inline void foo(int i) { printf("%d\n",i); }
```

type definitions

```
struct foo {
   int k; double d;
};
typedef struct foo bar[5];
```
#define FOO 10
#define BAR(X) int X[FOO];
BAR(goo); // means int goo[10];

#ifdef FOO
// this part is only interpreted if FOO is defined
#else
// this part is only interpreted if FOO is not defined
#endif

Example use of preprocessor directives:
   #define DEBUG 1
   ...
   ifdef DEBUG
       ... some awful debugging write...
   endif /* DEBUG */
Then
   gcc -DDEBUG=1 foo.c # includes debugging
   gcc -UDEBUG foo.c # doesn't include debugging
**const**: the thing I am declaring can't be changed.

**static**: the thing I am declaring is local to this file, but is a global variable in this context (which can be a function).

**extern**: the thing I am declaring is defined elsewhere.

```c
static int foo(int i) { return i*2; }
```

- In a `.c` file, creates something that is not exposed to the linker, and thus not available in other `c` files.
- In a `.h` file, creates a non-shared copy in every `.c` file that includes it. **This is allowed.**
A thing that is static is a global variable with a local scope. It applies to the variable being declared, and must be positioned before it.

In subroutines:

```c
int foo() {
    static int count=0; // global lifetime, local scope
    count++;
    return count;
}
```

returns the count of the number of times foo is called. The opposite of static is dynamic, i.e., created on the stack and deleted when the function ends.

in a file, outside a function:

```c
static int k; // only accessible in this file.
static void foo(); // don't let other files use it.
```

Rudeness with static: can expose via a pointer that is not static

```c
int *p = &k; // not static, so its contents are exposed.
```
Declare all variables static if possible -- limit scope.
Be polite about static and const: you can override them but don't.
extern allows one to specify that something is available elsewhere.

    extern int foo();  // someone else defined foo
    extern double timer;  // someone else defined timer

Used to give the type of something that will be found elsewhere (in another file)

In one file:
    int timer;
In another file
    extern double timer;

In one file:
    int timer;
In another file:
    extern int timer();

The compiler checks type conformality in each file, but not between files! -> need to use .h files so that in-file checks are enough.
Pointers to function

Function names are pointers to their addresses. Can declare a "pointer to function" that describes where a function is.
If bar is a function,
Then bar; // computes where the function is, and discards.
But bar(); // calls bar.

Examples
void bar() { printf("hi"); }
void pan(int i) { printf("%d\n",i); }
void (*foo)(); // foo is a pointer to a function of zero args
foo=bar; // no &: bar is already a pointer!
foo(); // exactly equivalent to bar()
foo = (void (*)(int)) pan; // ignore the arguments!
((void (*)(int))foo)(1); // call correctly via another cast!

^~~~~~~~ think of thing as a function of one argument: called a **method cast**.
foo is a function of 0 arguments.
override that (and waste space!)

Note: if a function is called with extra arguments, they are in general **ignored**.
If a function is called with fewer arguments, *unpredictable stack (and register) data* get interpreted as arguments (whether they are valid or not!)
Can declare a "pointer to function" that describes where a function is.

```cpp
void (*foo)(); // foo is a pointer to a function of zero args

```

Think of thing as a function of one argument: called a method cast.

Note: if a function is called with extra arguments, they are in general ignored. If a function is called with fewer arguments, unpredictable stack (and register) data get interpreted as arguments (whether they are valid or not!)

Types Page 18
Common programming problem
   int i;
   int *goo() { return &i; }
   ...
   if (goo!=NULL) { ... } /* ALWAYS true */
   if (goo()!=NULL) { ... } /* correct */

Perfectly legal
   printf("first byte of foo is %c\n",
         *((const char *)foo));
Example: device drivers: a really simple operating system might have:

```c
struct device_info {
    void *device_data;
    int (*write)(void *data, const char *buf, int size);
    int (*read)(void *data, char *buf, int size);
} devices[NDEVICES];
```

write is a **pointer to function** that writes something to a specific device.
read is a **pointer to function** that reads something from a specific device.

The kernel initializes this table so that write and read point to the actual functions.

So, to write "hello" to device n, you'd call
```
    devices[n].write(devices[n].device_data,"hello",5);
```
while to read 20 bytes from device n, you'd call
```
    char buf[20];
    devices[n].read(devices[n].device_data,buf,20);
```

Very important: there is a huge difference between
devices[n].reset;  // computes where a function is,
    // and then loses it "j;"

and

    devices[n].reset();

with the ()'s, it is a function call;
without the ()'s, it is a function pointer.
It gets really difficult, really fast, to notate complex C types. One can have an array of functions returning arrays of functions!

typedefs provide one way to simplify notation.
The basic idea of a typedef: define what is unknown in terms of what is known.

- Every typedef contains an "unknown symbol"
- The meaning of a typedef is a substitution of a symbol in a formula.

Example

```c
typedef int (*fint)();
unknown symbol: fint
usage:
fint x;
means
int (*x)();
```

typedef caveats

- can only have **one unknown symbol**.
- if the symbol has a known type, the typedef has a **syntax error**.
- repeating a typedef twice **always** causes a compiler error.
typedefs are "like" classes in C++, but they are not identical to classes.

- Problem is with "type equivalence"
  - If two classes have different names, C++ considers them to be **differing types**.
  - If two typedefs resolve to the same base definition, C considers them to be the **same type**.

Example:
```c
typedef long pid_t;
typedef pid_t bigarr[1000];
bigarr x;
long y[1000];
```
Then x and y are considered to have the same type, regardless of how that type is defined!
Types and function prototypes

- In C++, functions with the same name and different arguments are considered to be **different**.
- In C, **the function name defines the function** and is independent of the types of arguments.
- It is possible to define a single function that takes many types of arguments (e.g., printf).

C function calling

- the function's prototype (argument types) determines how data in the stack frame will be referenced, but not its actual content.
- The function's call determines how the data in the stack frame will be laid out in memory.
- If these differ, havoc can result!
General header convention:

- Header files containing typedefs can't be included twice or havoc results.
- Why? Because typedefs define the unknown symbols in them. It is an error to define something twice.
- General trick:

```c
#ifndef STDIO_H_DEFINED
#define STDIO_H_DEFINED 1
... rest of header ...
#endif /* STDIO_H_DEFINED */
```

If a header file consists of this code, then including it twice has no effect. Almost all header files are "proofed" against being included twice in this way.

```c
#ifdef X == #if defined(X)
```
The "why" of typedefs

Naively, typedefs reduce the length of a declaration:
  typedef struct _IO_FILE FILE;
means that you can type:
    FILE *s;
rather than
    struct _IO_FILE *s;

But there is another use of typedefs: portability.
Suppose you want something to run on both ia32 and
ia64.
    On an ia32, ints are 32 bits long.
    On an ia64, ints are 64 bits long.
But you want something that doesn't change in size.

So, we write (on ia64):
  typedef long int __clock_t; // define a base type
  typedef __clock_t clock_t;  // define an exposed type

Why so many layers?
  Each layer addresses a different form of portability
  First layer __clock_t: the kernel's idea of time.
  Second layer clock_t: the application's idea of time.

You will almost never see int in an OS.
You will see int32_t, int64_t, uint32_t, and uint64_t.
The declaration of sigaction is:

```c
int sigaction(int signum, const struct sigaction *act,
              struct sigaction *oldact);
```

Where

```c
struct sigaction {
  void     (*sa_handler)(int);
  void     (*sa_sigaction)(int, siginfo_t *, void *);
  sigset_t   sa_mask;
  int        sa_flags;
  void     (*sa_restorer)(void);
};
```

This says that one can define an action via either

```c
void my_sigaction(int signal, siginfo_t *info, void *data) {
  ...
}
```

or

```c
void my_handler(int signal) {
  ...
}
```

(but not both).

(In most architectures, you can't assign both to sigaction and handler; they're stored as a union)

The extra arguments of a sigaction:

- info: detailed information about the signal.
- data: a ucontext_t object (if available), which represents the execution context in which the signal occurred.

Very subtle question: why is data of type void *?

Answer: different systems define this information differently.
To create a thread, one has to jump through some rather complex hoops:

Define the data
to be passed to the thread.
that the thread returns when done.
Both of these types are determined by the thread.

So, to create a thread, we use the very cryptic:

```c
int pthread_create(
    pthread_t *thread,
    const pthread_attr_t *attr,
    void *(*start_routine) (void *),
    void *arg
);
```

where

- **thread** is the thread descriptor
- **attr** describes how the thread should behave
- **start_routine** is the routine to call
  - takes "arg" as argument, returns a return type.
- **arg** is the argument to send to **start_routine** (thread input)

Some mysteries:

Q: why so many "void *" declarations?
A: the actual types are determined by the thread and its containing process, not the operating system.
Q: why return a pointer rather than an object?
A: Pointers take constant space; objects take variable space on the stack!

http://www.cs.tufts.edu/comp/111/examples/Threads/pthreads1.c