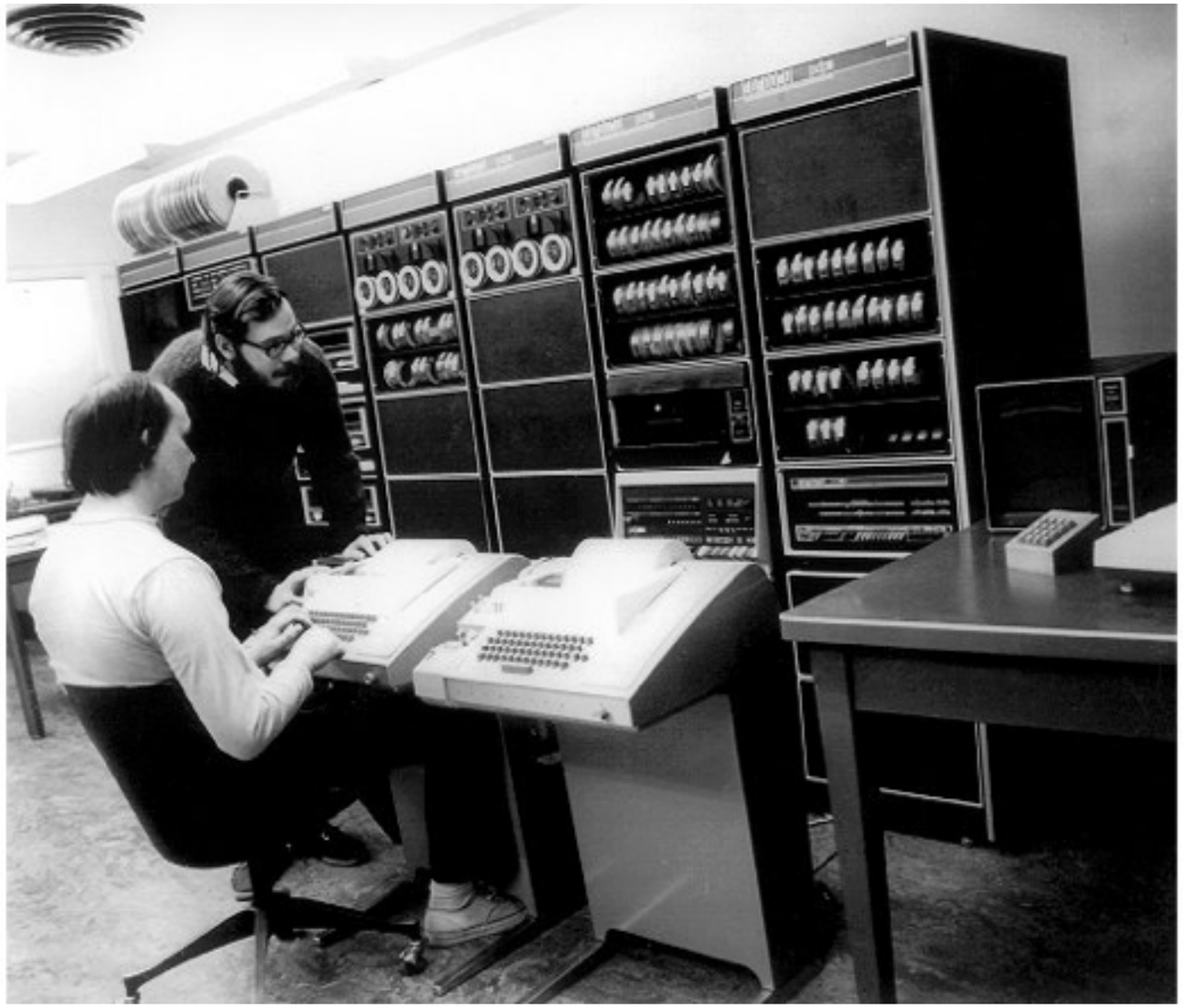
Program Structure

"to get a deeper understanding of the language"



Deep C - a 3 day course

Jon Jagger & Olve Maudal

header files

can contain

- #includes
- macro guards (idempotent)
- macros (e.g. EOF)
- type declarations (e.g. FILE)
- external data declarations (e.g. stdin)
- external function declarations (e.g. printf)
- inline function definitions

should not contain

- function definitions (unless inline)
- data definitions

header file example

#IF Not DEFined

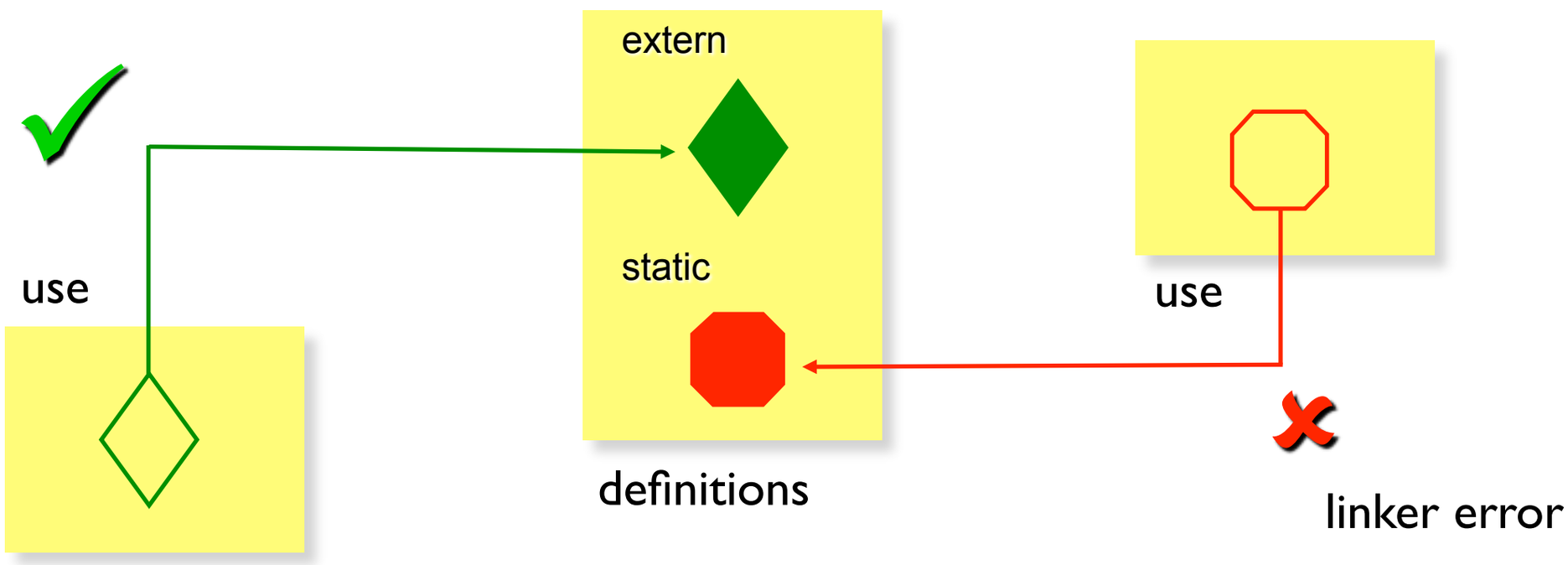
ALL UPPERCASE
is a very strong preprocessor convention

stdio.h

```
#ifndef STDIO_INCLUDED
#define STDIO_INCLUDED

#define EOF    ...

typedef struct FILE FILE;

extern FILE * stdin;

FILE * fopen(const char * path, const char * mode);
int fflush(FILE * stream);
...
void perror(const char * diagnostic);
FILE * tmpfile(void);
...

#endif
```

linking

- a linker links the use of an identifier in one file with its definition in another file
- an identifier is made available to the linker by giving it external linkage (the default) using the `extern` keyword
- an identifier is hidden from the linker by giving it internal linkage using the `static` keyword



function declaration linkage

- default to external linkage
- extern keyword makes default explicit

time.h

```
struct tm * localtime(const time_t * when);
time_t time(time_t * when);
```

equivalent

time.h

```
extern struct tm * localtime(const time_t * when);
extern time_t time(time_t * when);
```

function definition linkage

- default to external linkage
- use static keyword for internal linkage

time.c

```
time_t time(time_t * when)
{    ...
}
```

equivalent

time.c

```
extern time_t time(time_t * when)
{    ...
}
```

source.c

```
static void hidden(time_t * when);

static void hidden(time_t * when)
{
    ...
}
```

data linkage

- without a storage class or an initializer
- the definition is tentative – and can be repeated
- this is confusing and not compatible with C++

ok in C, duplicate definition errors in C++

```
int v; // external, tentative definition
...
int v; // external, tentative definition
```

recommendation: extern data *declarations*

- use explicit extern keyword, do not initialize

recommendation: extern data *definitions*

- do not use extern keyword, do initialize

multiple declarations ok

```
extern int v;
extern int v;
```

single definition with initializer

```
int v = 42;
```

; as separators -> sequence point, heartbeat
{ blocks } as grouping mechanism

-Wmissing-prototypes

opaque types

forward declarations - when are they ok

ensure header files are self-contained
  include own header first
  compile headers as part of build?

ensure header files are idempotent
  macro guards

TODO?: show (*p) - (*q)++ example again to motivate restrict

restrict

- applies only to pointer declarations
- `type*restrict p` → `*p` is accessed only via `p` in the surrounding block
- enables pointer no-alias optimizations
- a compiler is free to ignore it

c99

```
void f(int n, int * restrict p, int * restrict q)
{
    while (n-- > 0) {
        *p++ = *q++;
    }
}
```

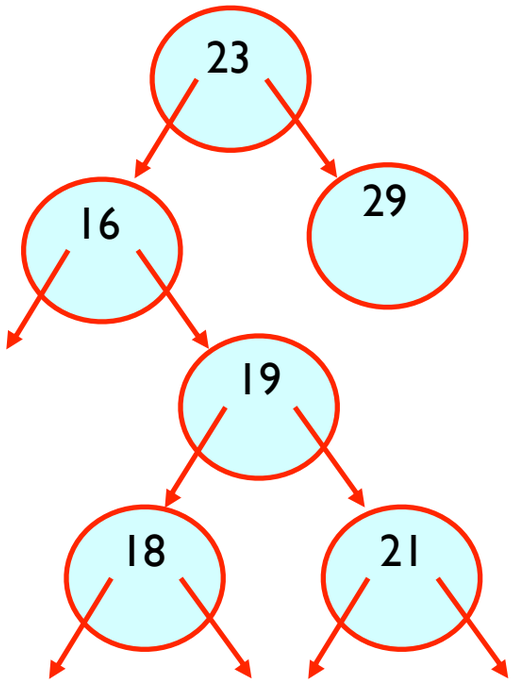
```
void g(void)
{
    int d[100];
    f(50, d + 50, d);  // ok
    f(50, d + 1,  d);  // undefined-behaviour
}
```

restrict

c99

- the restrict can also be used on struct pointer members
- enables the same no-alias optimizations

```
struct tree_node
{
    int count;
    struct tree_node * restrict left;
    struct tree_node * restrict right;
};
```



blocks / compound statement

- a compound statement (aka a block) is:
- an unnamed sequence of statements and declarations
- grouped together inside { braces }

```
;
```

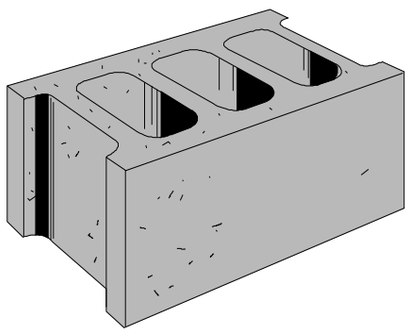
the null statement

```
declaration;

expression;
```

semicolons required

```
{
    ...  ;
    ...  ;
    ...  ;
}
```

trailing semicolon not required



type linkage?

- `static` can be used on a type definition
- it has no effect
- in C type names do not have linkage (they do in C++)
- don't do it

?

```
static struct date
{
    int year, month, day;
};
```

?

```
static enum month { january, ... };
```

todo

- enums cannot be forward declared?
- creates bad dependencies
- wrap single data member in a struct and look at assembler - no overhead