Functions and Program Structure

fn declarations

- a promise that a function exists
  - typically written in header files
  - return type must be explicit
  - use void for no parameters and no return value

stdio.h

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

parameter names are optional but help readability and documentation

1. f(int a, int b) cannot be shortened to f(int a, b);
2. avoid bool parameters as they can be confusing on their own

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 example

IF Not DEFined

stdio.h

all uppercase: a strong convention

```
#ifndef STDIO_INCLUDED
#define STDIO_INCLUDED

#define EOF    (-1)

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
```

- a definition honours the promise
  - written in source files

parameter names are required

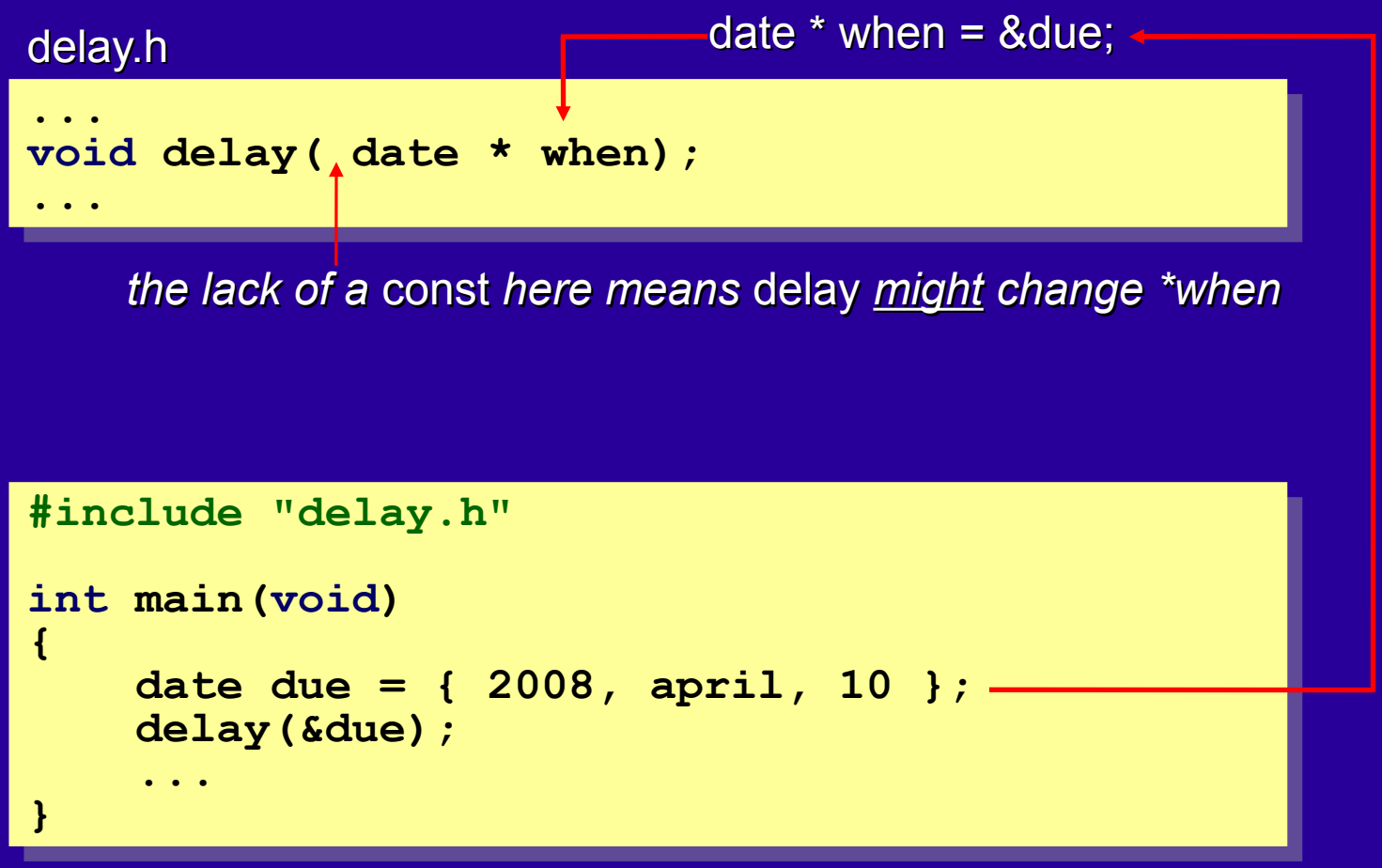
stdio.c

```
#include <stdio.h>
...
FILE * fopen(const char * path, const char * mode)
{    ...
}

int fflush(FILE * stream)
{    ...
}

void perror(const char * message)
{    ...
}
...
```

parameter name can be different to that used in the function declaration

pass by pointer

- use a pointer to a non-const
  - if the definition needs to change the target



- changing the parameter does not change the argument

size_t from = 0;

size_t to = size;

```
bool search(
    const int values[], size_t from, size_t to,
    int find)
{
    while (from != to && values[from] != find)
    {
        from++;
    }
    return from != to;
}
```

```
int main(void)
{
    ...
    ... search(array, 0, size, 42);
    ...
}
```

- works for enums and structs too
  - but not for arrays (unless the array is inside a struct)

date.h

date when = today;

```
...
const char * dayname(date when);
...
```

```
#include "date.h"
...
int main(void)
{
    date today = { 2008, april, 10 };
    ...
    puts(dayname(today));

    assert(today.year == 2008);
    assert(today.month == april);
    assert(today.day == 10);
}
```

Thursday

ptr to const

- an efficient alternative to pass by copy
  - except that the parameter can be null

date.h

const date * when = &today;

```
...
const char * dayname(const date * when);
...
```

the const *here promises that* dayname *wont change *when*

```
#include "date.h"
...
int main(void)
{
    date today = { 2008, april, 10 };
    ...
    puts(dayname(&today));

    assert(today.year == 2008);
    assert(today.month == april);
    assert(today.day == 10);
}
```

Thursday

parameter advice

- pass by plain pointer...
  - when the function needs to change the argument
- pass by copy for built in types and enum...
  - they are small and they will stay small
  - copying is supported at a low level
  - very fast
- for most structs...
  - mimic pass by copy with pointer to const
  - they are not small and they only get bigger!
  - very fast to pass, but be aware of indirection cost
- for some structs...
  - pass by copy can be OK, occasional optimization

Parameter order

- list output parameters first
  - loosely mimics assignment

```
char * strcpy(char * dst, const char * src);
```

```
#include <string.h>

...
{
    const char * from = "Hello";
    char to[128];
    ...
    //      to = from
    strcpy(to , from);
}
```

top level const?

makes no sense on a function declaration

```
const char * day_suffix(const int days);
```

the const *here is meaningless*
write this instead

```
const char * day_suffix(int days);
```

the const *here (on a function definition) is reasonable and states that the function will not change* days

```
const char * day_suffix(const int days)
{
    ...
}
```

register variables

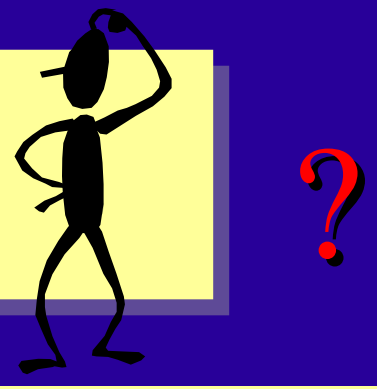
- a speed optimization hint to the compiler
  - compiler will use registers as best it can anyway
  - effect is implementation defined
  - register variables can't have their address taken

```
void send(register short * to,
          register short * from,
          register int count)
{
    register int n = (count + 7) / 8;
    switch (count % 8)
    {
    case 0 : do {   *to++ = *from++;
    case 7 :        *to++ = *from++;
    case 6 :        *to++ = *from++;
    case 5 :        *to++ = *from++;
    case 4 :        *to++ = *from++;
    case 3 :        *to++ = *from++;
    case 2 :        *to++ = *from++;
    case 1 :        *to++ = *from++;
                } while (--n > 0);
    }
}
```

statics

- a local variable with static storage class
  - a local variable with 'infinite' lifetime
  - best avoided – subtle and hurts thread safety
  - but ok for naming magic numbers (as are enums)

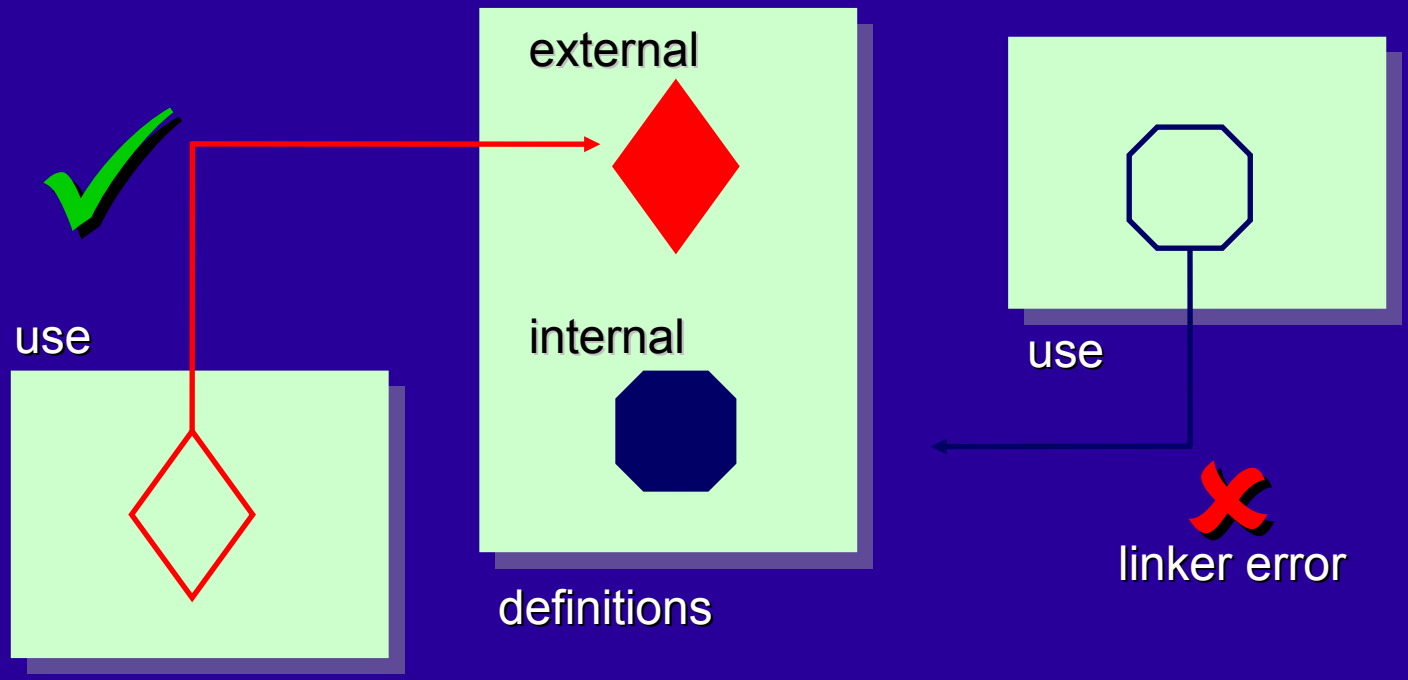
```
int remembers(void)
{
    static int count = 0;
    return ++count;
}
```



```
void send(short * to, short * from, int count)
{
    static const int unrolled = 8;

    int n = (count + unrolled - 1) / unrolled;
    switch (count % unrolled)
    {
        ...
    }
}
```

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



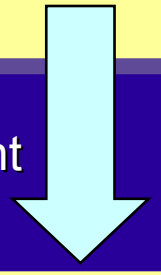
function declarations

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

time.c

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

time.c

equivalent

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

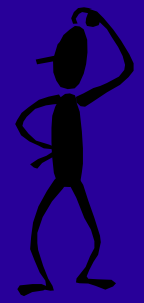
source.c

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

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

- 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 ISO 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

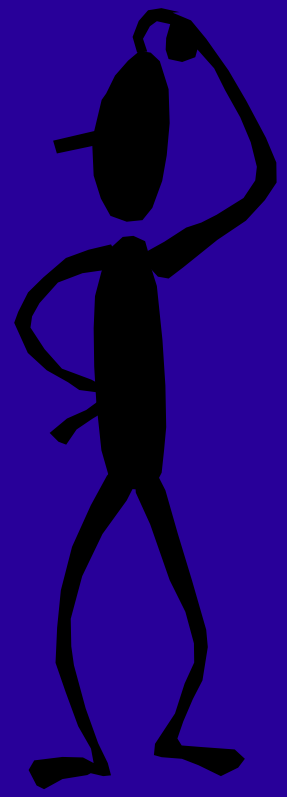
```
int v = 42;
```

type linkage?

- static can be used on a type definition
  - it has *no* affect - type names do not have linkage
  - don't do it

```
static struct wibble
{
    ...
};


static enum fubar { ... };
```

??



- inline function rules
  - all declarations must be declared inline
  - there must be a definition in the translation unit – a file-scope declaration with extern is a definition
  - does not affect sequence point model – there is still a sequence point before a call to an inline function

is_even.h

```
#include <stdbool.h>

static inline bool is_even(int value)
{
    return value % 2 == 0;
}
```

__func__

- the name of the current function is available
  - via the reserved identifier __func__

```
void some_function(void)
{
    puts(__func__);
}
```

c99

as-if compiler translation

```
void some_function(void)
{
    static const char __func__[] =
        "some_function";
    puts(__func__);
}
```

functions with a variable no. of arguments

- helpers in <stdarg.h> provide type-unsafe access

```
int printf(const char * format, ...);
```

```
#include <stdarg.h>

int my_printf(const char * format, ...)
{
  va_list args;
  va_start(args, format);
  for (size_t at = 0; format[at] != '\0'; at++)
  {
    switch (format[at])
    {
      case 'd': case 'i':
        print_int(va_arg(args, int)); break;
      case 'f': case 'F':
        print_double(va_arg(args, double)); break;
      ...
    }
  }
  va_end(args);
}
```

ptr to functions

- in an expression the name of function "decays" into a pointer to the function
  - ( ) is a binary operator with very high precedence
  - f(a,b,c) → f () {a,b,c}
- you can name a function without calling it!
  - the result is a strongly typed function pointer

* needed here

```
#include <stdio.h>

int add(int a, int b) { return a + b; }
int sub(int a, int b) { return a - b; }

int main(int argc, char * argv[])
{
    int (*f)(int,int) =
        argc % 2 == 0 ? add : sub;

    printf("%d\n", f(4, 2));
}
```

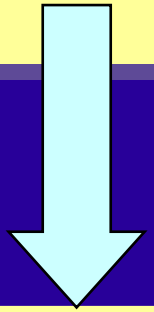
- function pointers can be function parameters!
  - * is optional on the parameter

```
#include <stdio.h>

int add(int a, int b) { return a + b; }
int sub(int a, int b) { return a - b; }

int call(int f(int,int))
{
    return f(4, 2);
}

int main(int argc, char * argv[])
{
    int (*f)(int,int) =
        argc % 2 == 0 ? add : sub;

    printf("%d\n", call(f));
}
```

```
int call(int (*f)(int,int))
{
    return (*f)(4, 2);
}
```

- typedef can help

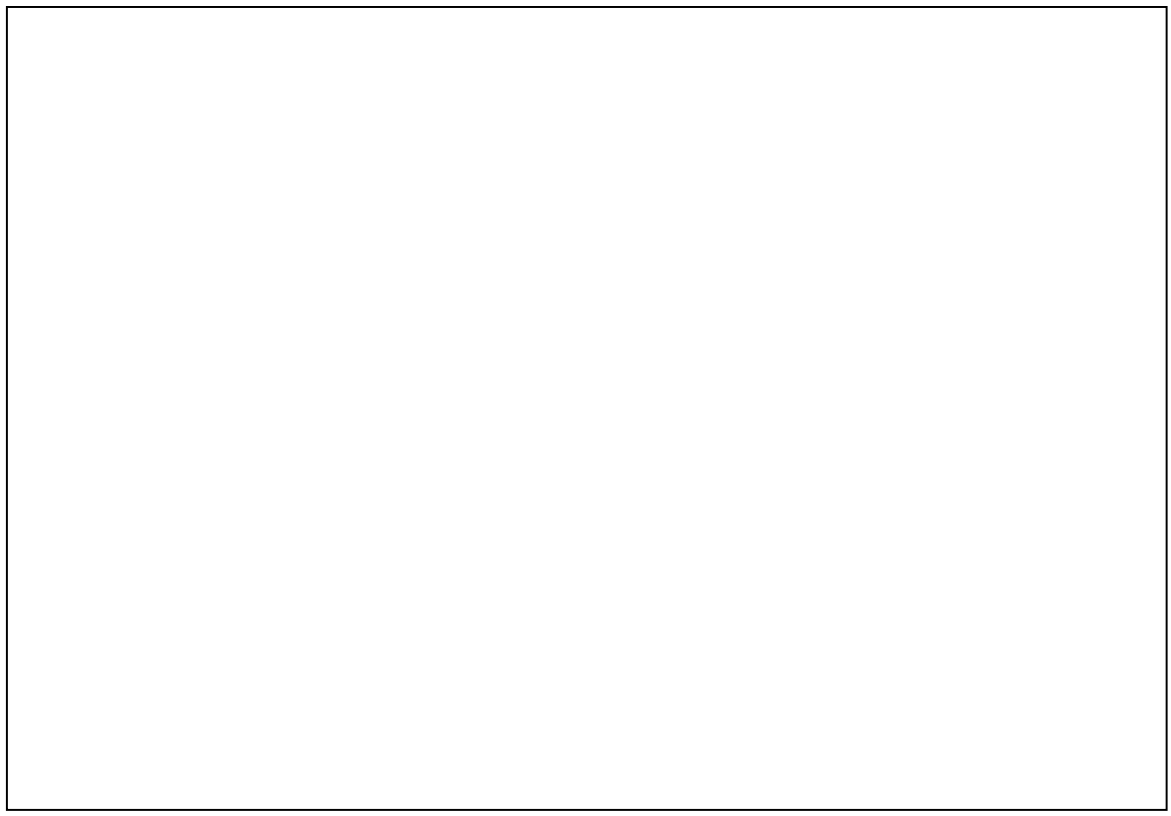
```
int call(int f(int,int))
{
    return f(4, 2);
}
```

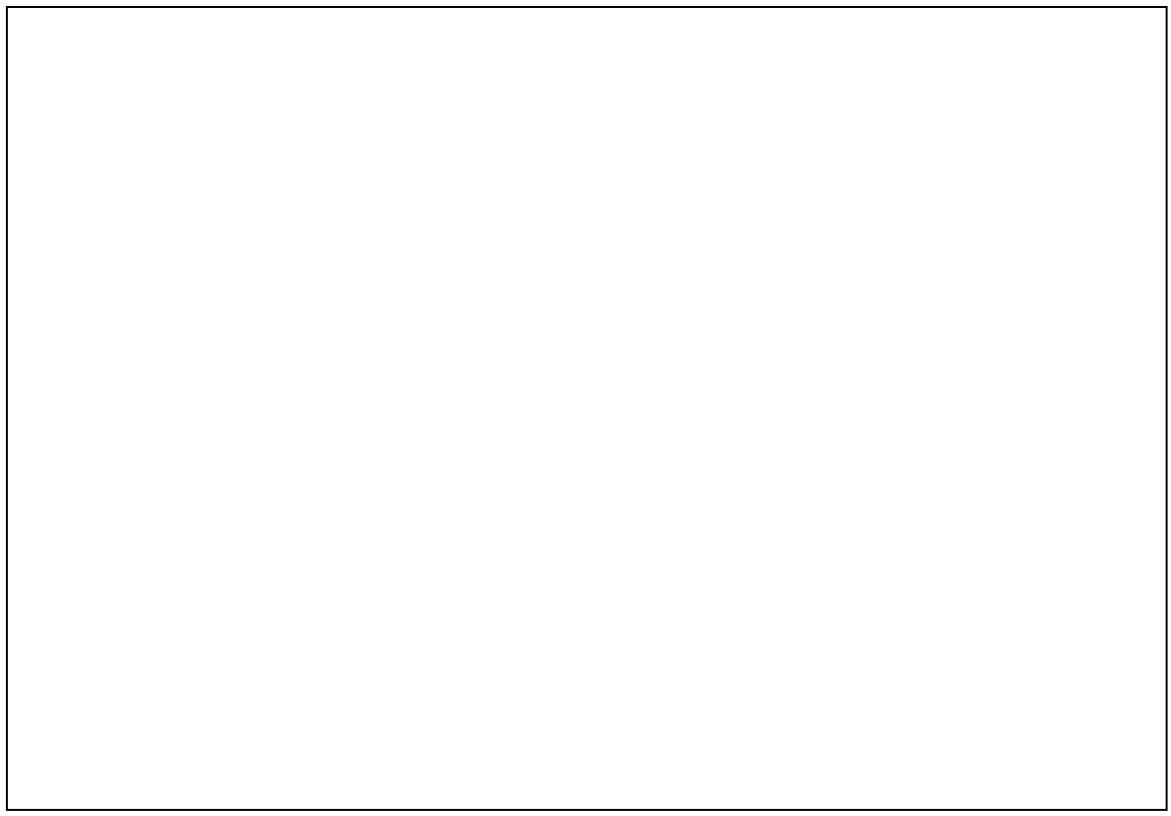


```
typedef int func(int, int);

int call(func * f)
{
    return (*f)(4, 2);
}
```

Summary

- ensure headers are idempotent
- don't define data in a header file
- mimic pass by copy using pointer to const for structs
- don't use the auto keyword
- don't use the register keyword
- use static local variables only if const
- give static linkage to anything in a source file not declared in its header





A function declaration can be declared anywhere (eg inside a block) but is typically written in a header file which is then #included. This avoids duplication and the obvious problems of maintenance should the declaration change.

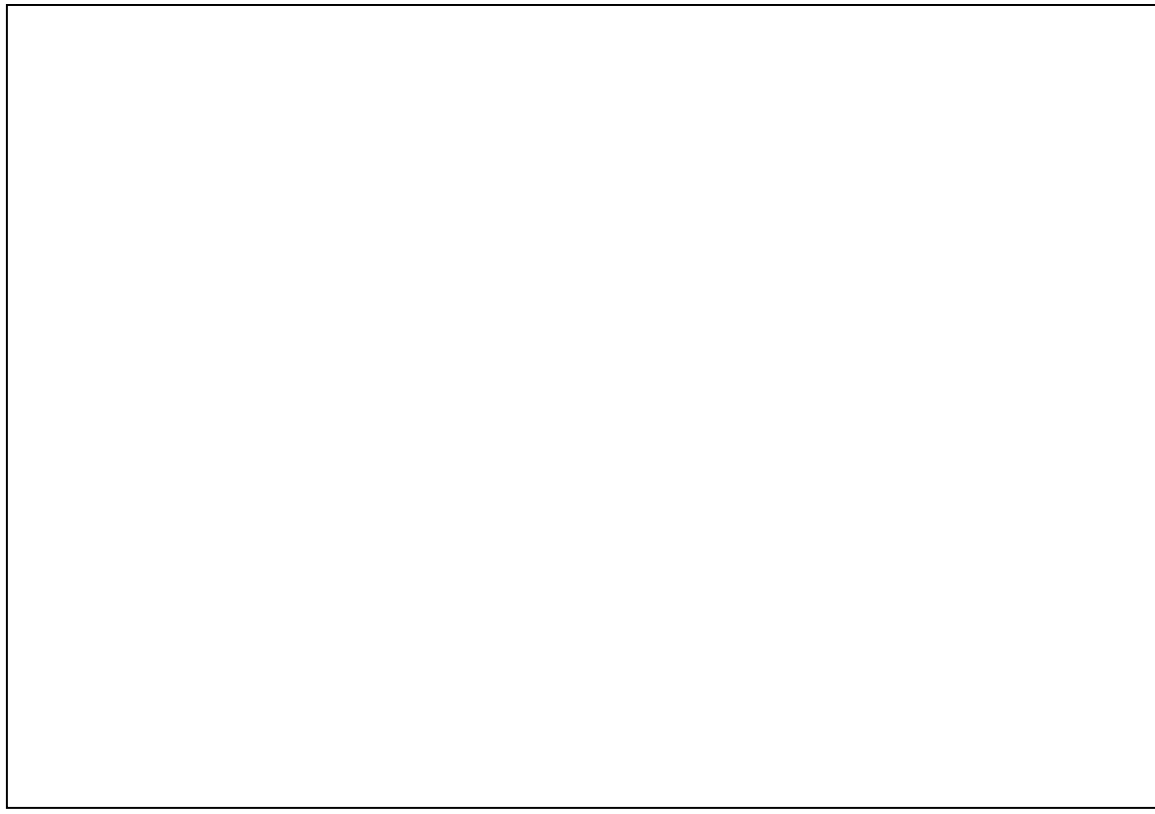
Before C99 function declarations could be written without a return type and would default to int (in fact there were several places in the syntax where a missing type would be assumed to be an int). C99 requires the return type to be explicitly stated.

The standard does not actually require a physical header file to exist for the standard header files such as stdio.h.

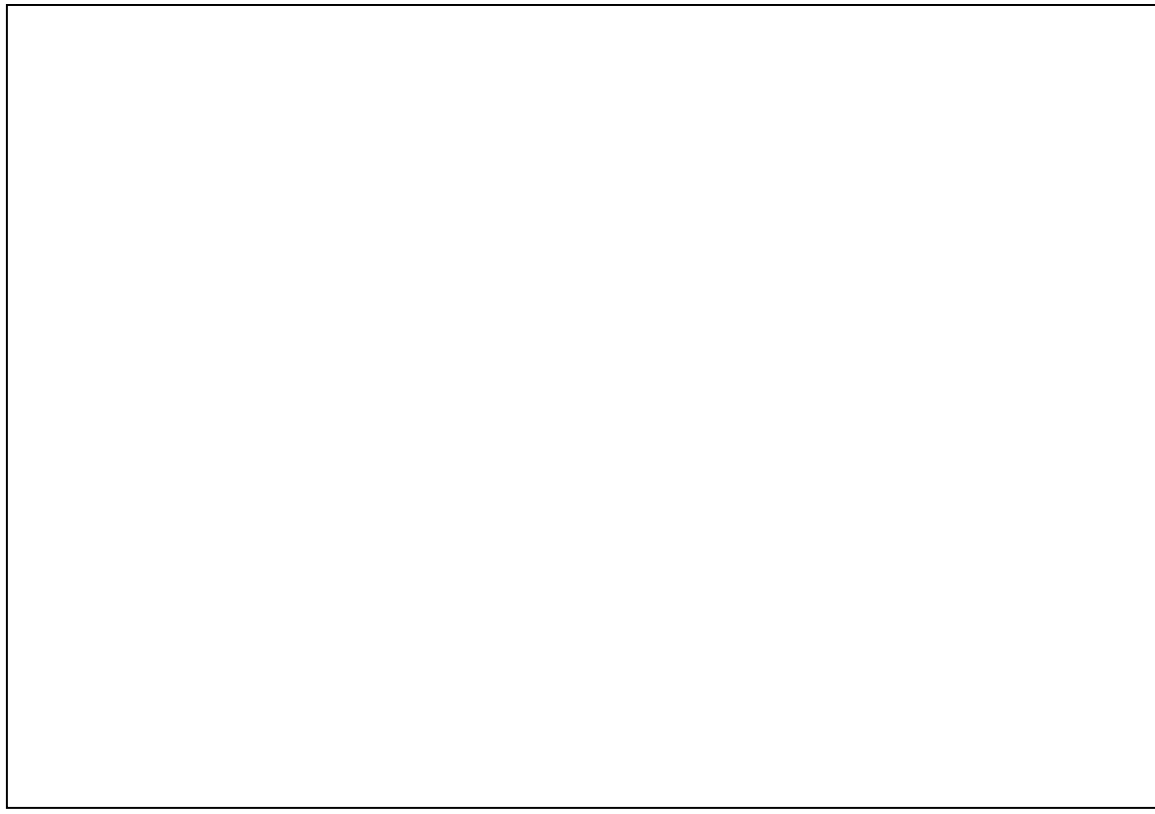
Normally in C you can declare multiple variables in a single declaration (eg int a,b;). However this syntax does not work for function parameters. The reason for this is the potential for ambiguity – consider that b could be an unnamed parameter of type b.

Functions with bool parameters invite true/false literals as arguments and thus foster unreadable code. They are also dangerous since so many types have an implicit conversion to bool – for example, if you accidentally pass a pointer to a function expecting a bool you probably won't even get a warning.

Function declarations are also known as function prototypes.



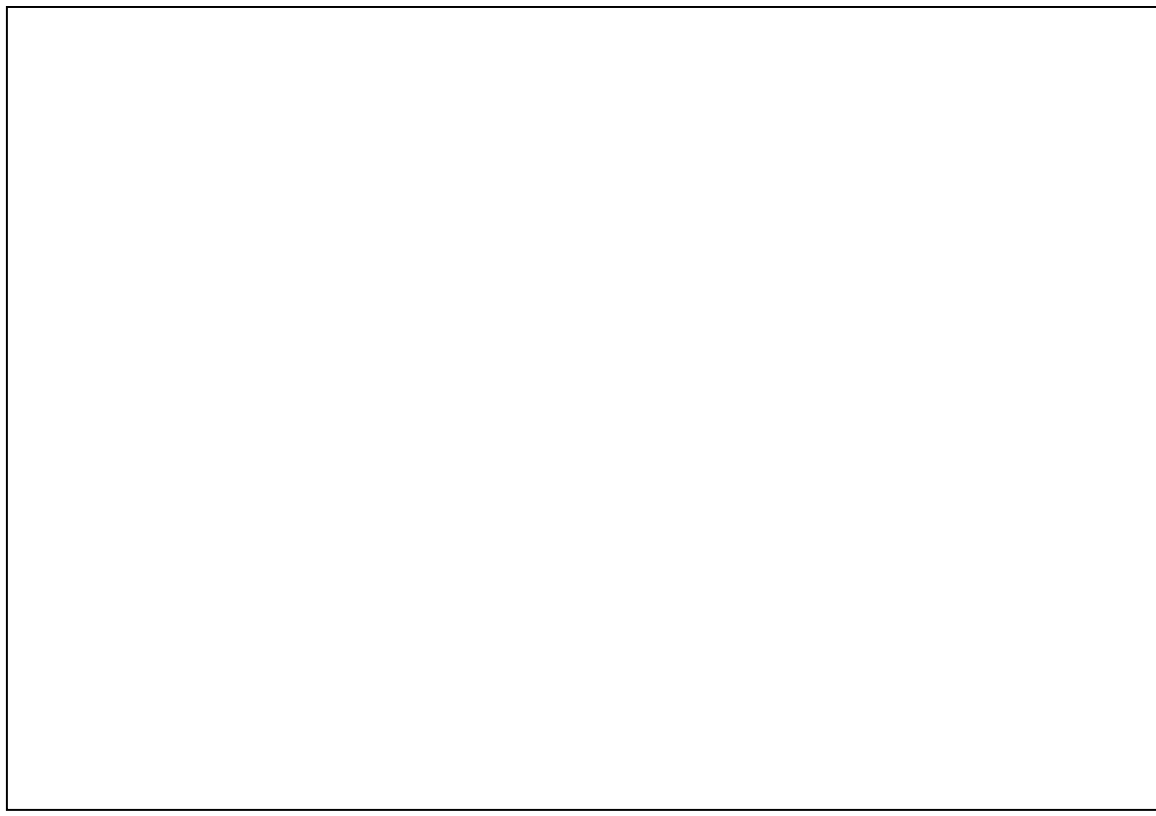
It is a good idea to create a standardized ordering for the declarations that occur in a header file. For example, the order specified on the slide, macro guards (if present) first etc. It is also worth standardising the order of #includes. For example, local header files before system header files, and in alphabetic order within each section.



Note that this is simply an example. The contents of <stdio.h> (if it exists at all) may be different on your particular machine.

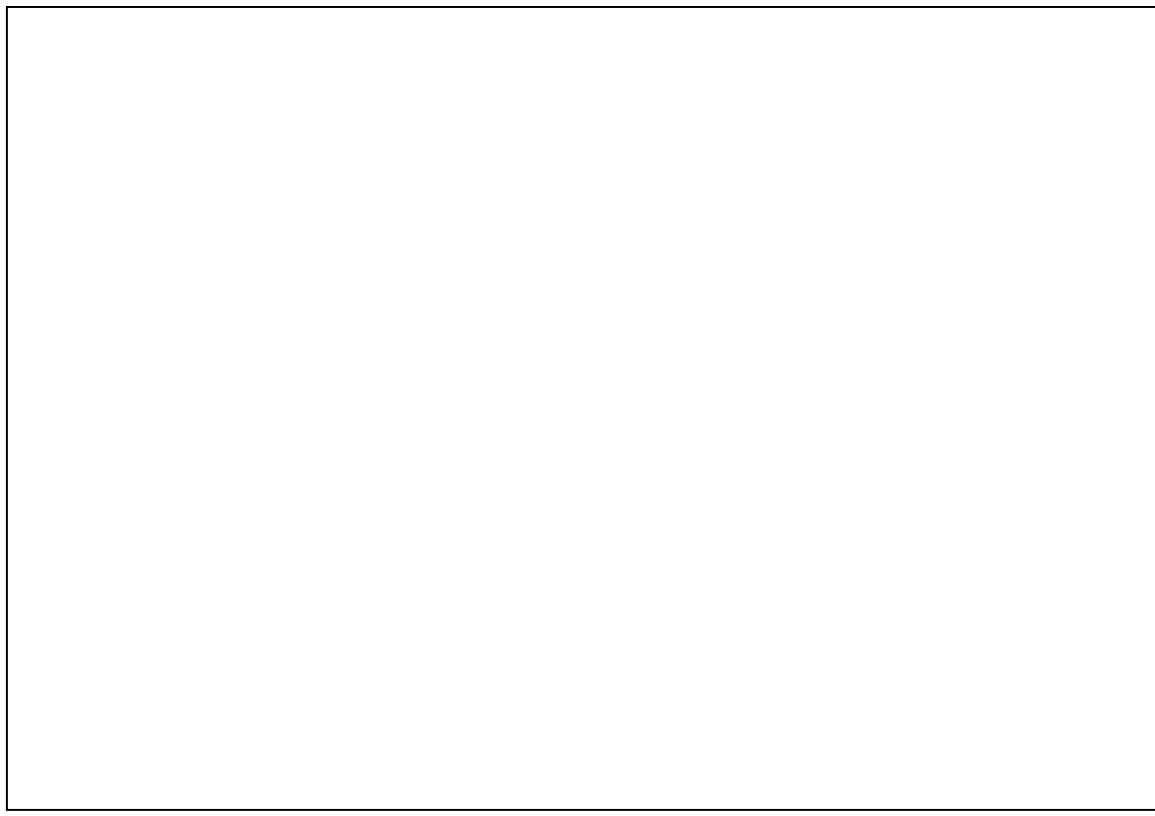
Macro guards are particularly important to avoid duplicate type definitions.

Note that all uppercase identifiers usually indicate a preprocessor token – the type FILE breaks this convention, an unfortunate accident of history and not something to be copied.

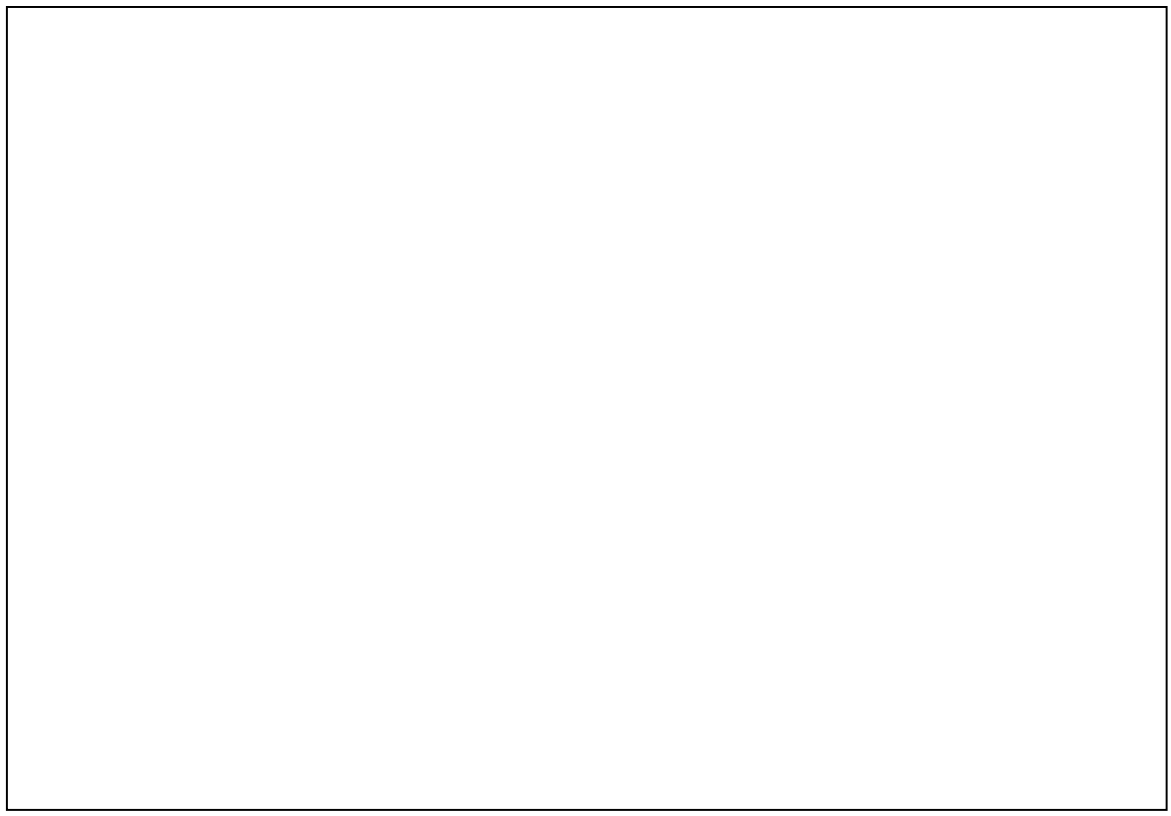


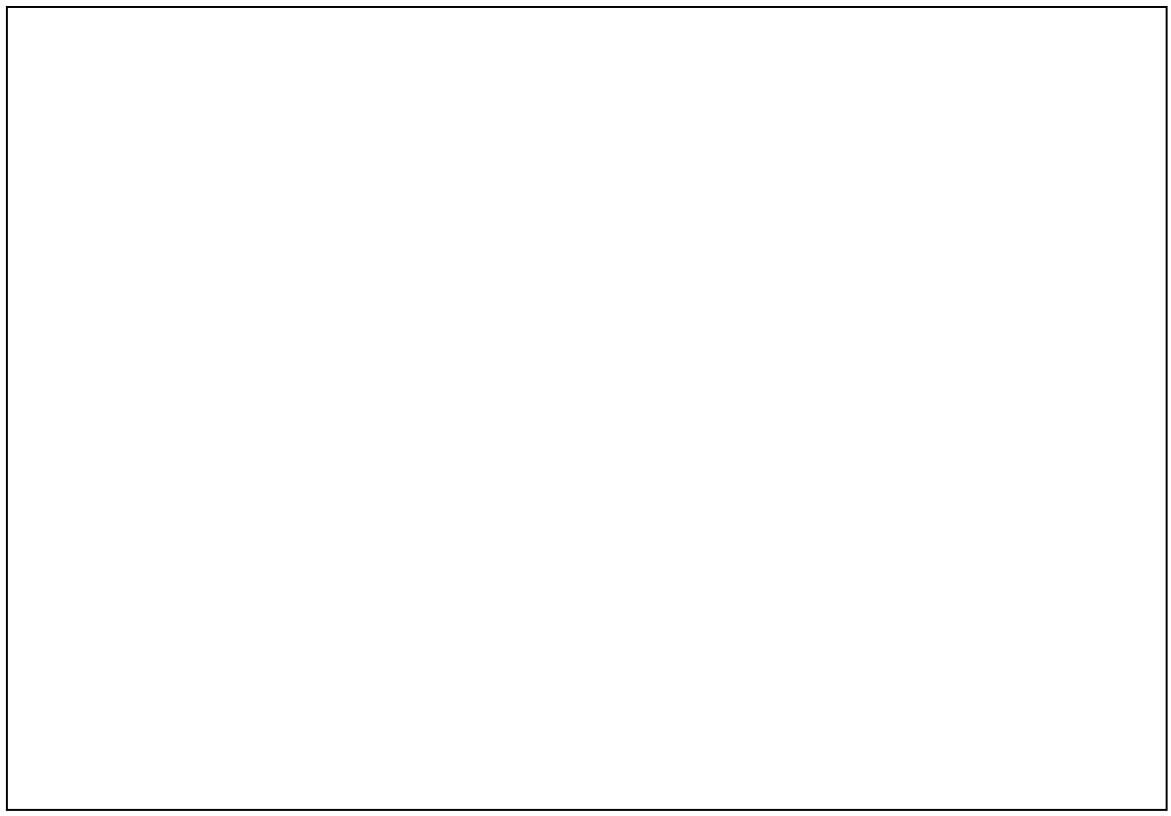
If a call is made against a function declaration then a matching function definition must be provided otherwise you will get a linker error.

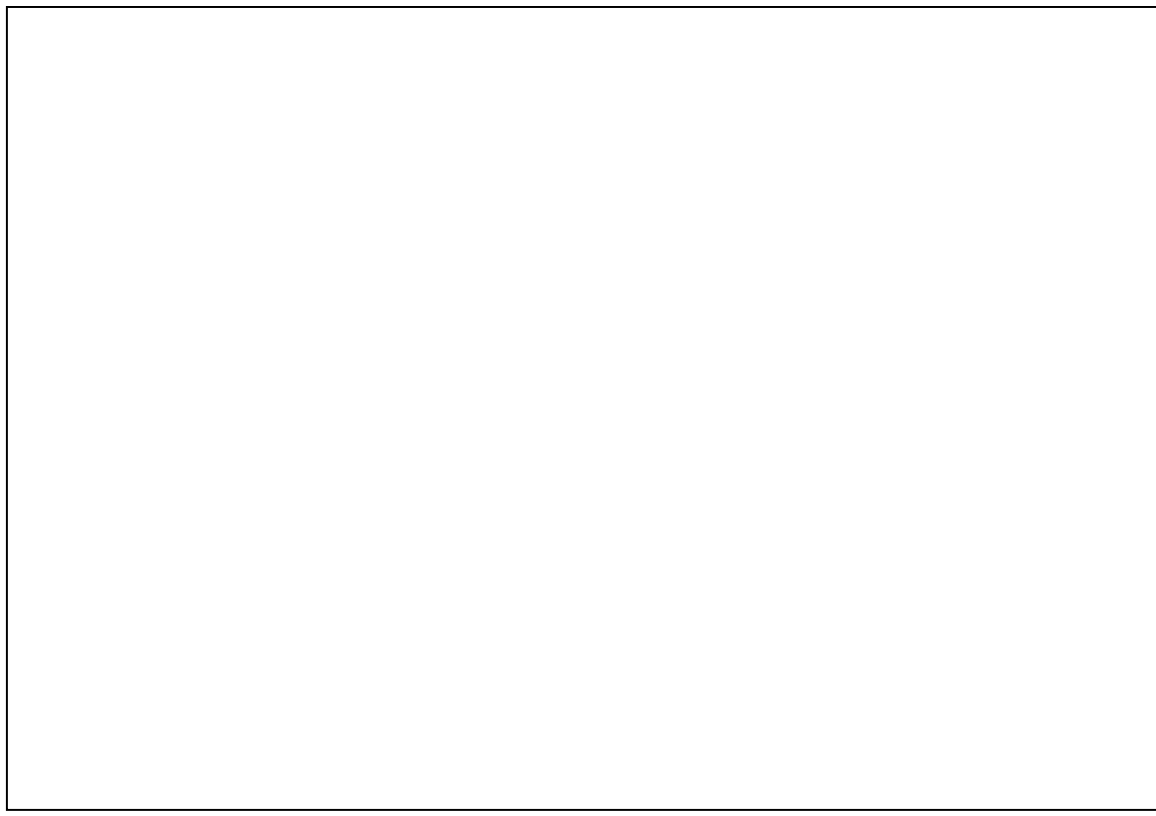
You cannot write a function definition inside another function definition.



Note that if a function parameter is declared as a pointer to a non-const then you *cannot* assume the function definition will definitely change the pointed to object. The lack of a const merely allows the function definition to change the pointed to object. Of course, if the function definition *never* changes the pointed to object then hopefully the signature should state this with a const.

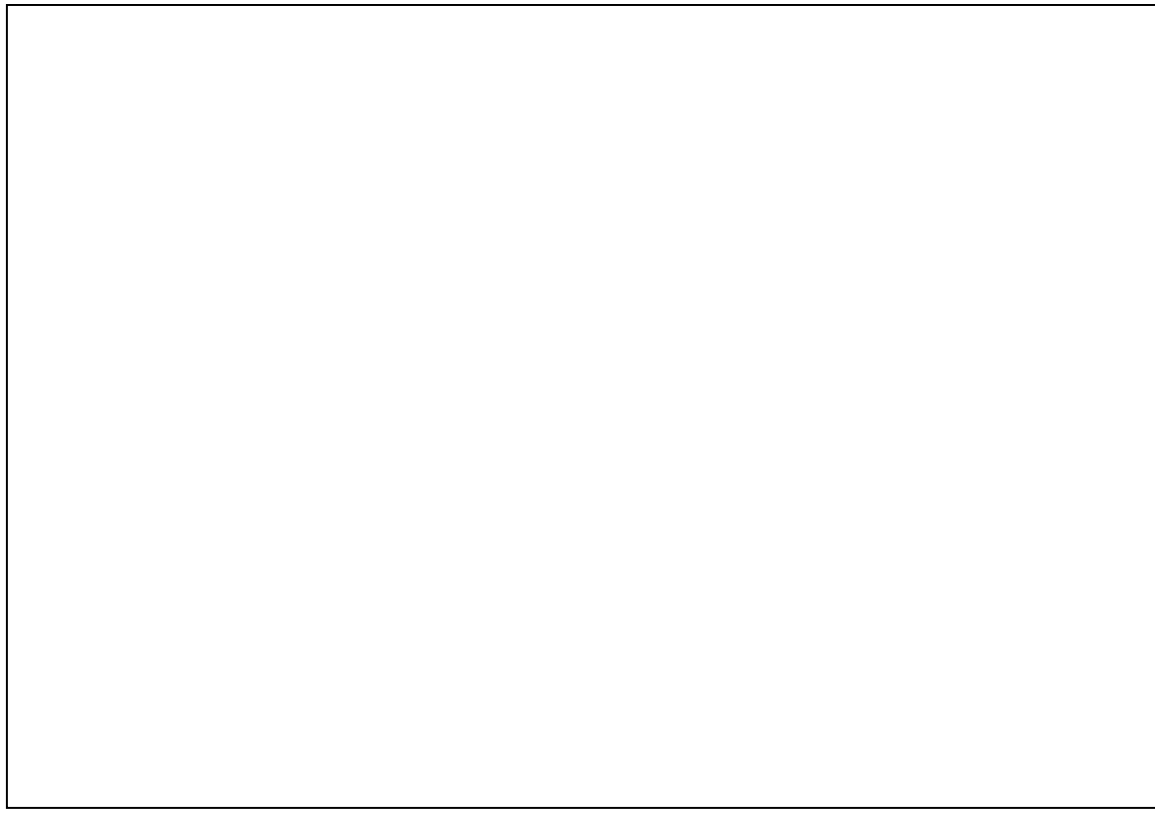




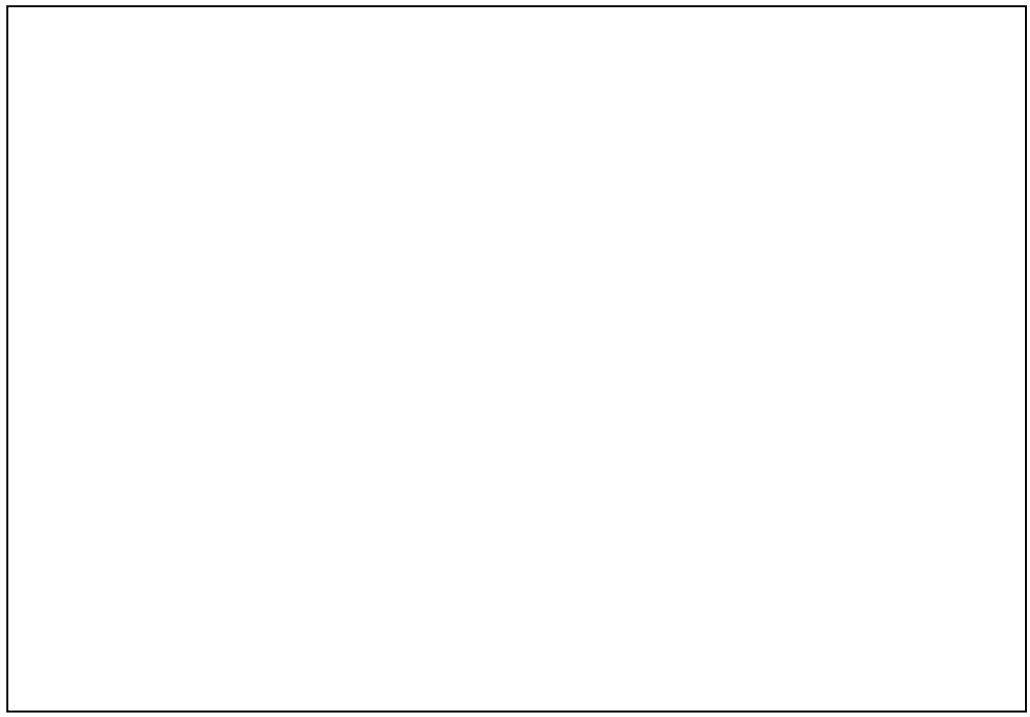


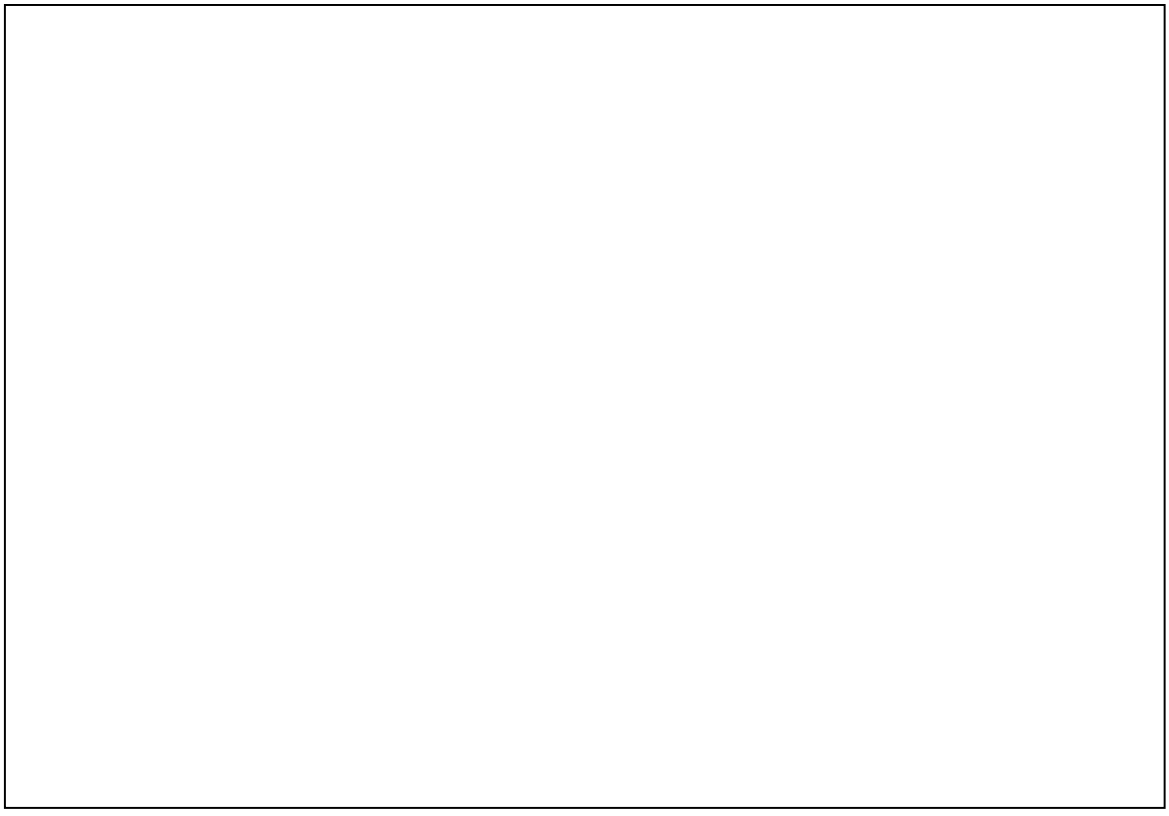
Of course, the function definition could cast away the constness and change the pointed object! Few things are guaranteed! However, you have to write code with some assumptions and it is better to write code well on the assumption that the code you are calling is also well written rather than the reverse.

Any function parameter that is a pointer could of course be null. If a function definition attempts to dereference a null pointer the result is undefined behaviour. All bets are off!



It is a good idea to create a standardized ordering for the declarations that occur in a header file. For example, the order specified on the slide, macro guards (if present) first etc.



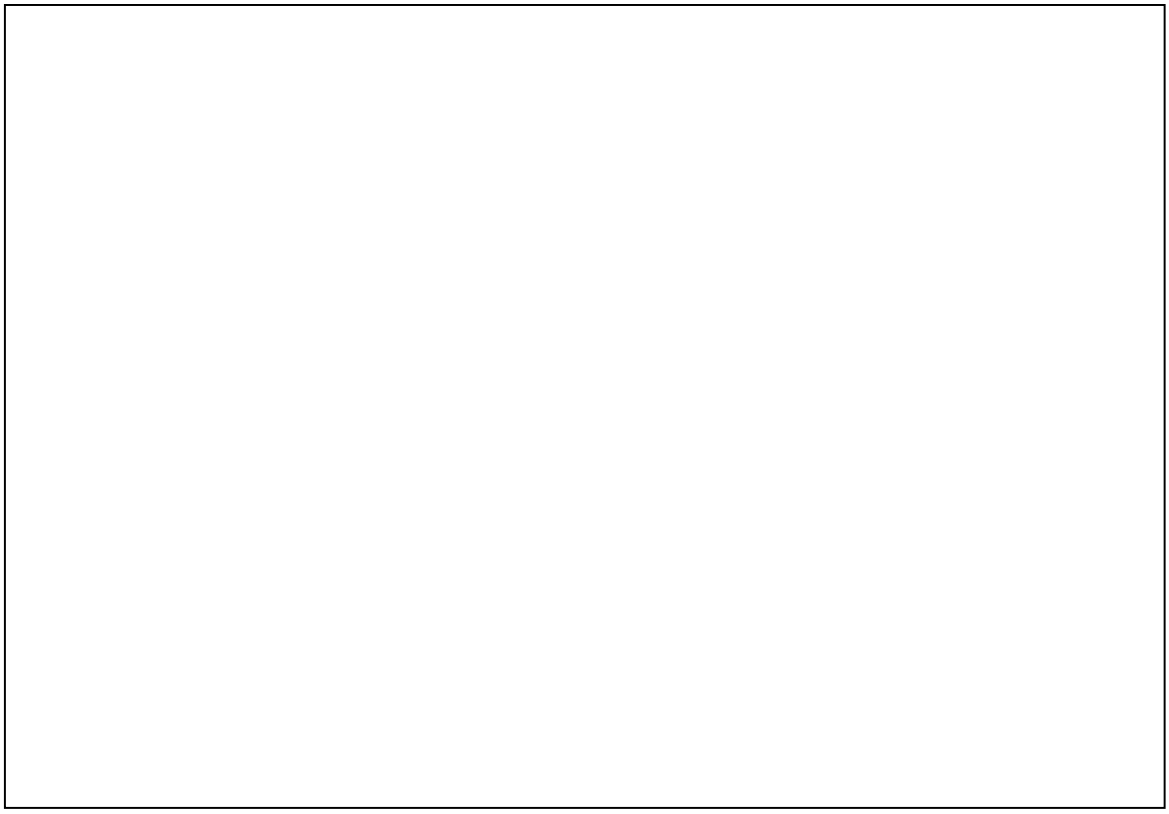


Using a const modifier on the top level of a parameter does not make sense. For example in fflush(FILE * const stream); the const indicates that stream cannot be modified. But what does this mean? It does not mean the caller's argument since stream is a copy of that. It does not mean stream in the definition of the fflush function since this is its declaration. Note however that in a function *definition* a top level const does make sense. Note also that the linker will happily bind a definition with a top level const as the definition of an equivalent declaration without the top level const.

Note however that C++ is different. C++ allows overloading so in C++ the following two declarations declare different functions:

```
extern void f(int);
extern void f(const int);
```

In C they declare the same function (it is just that the later says a bit more than the former).



Like const, register makes sense for function definitions but not for function declarations.

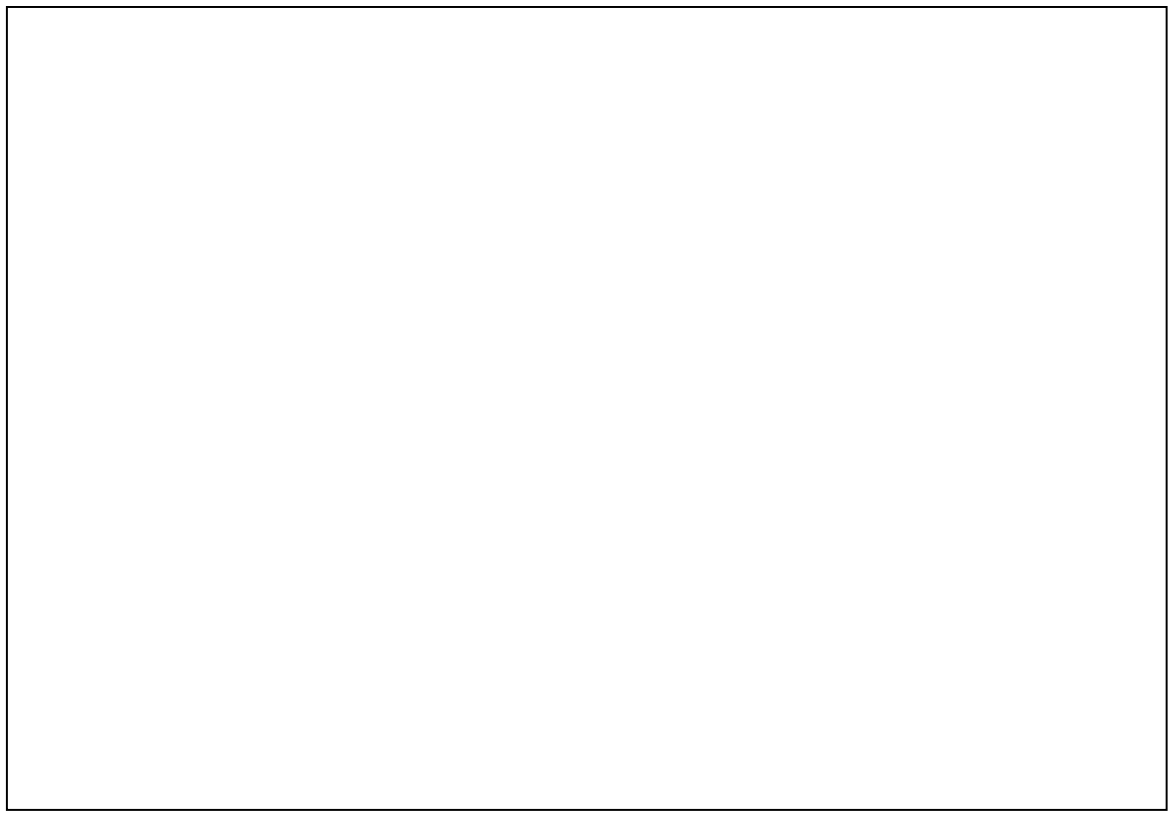
The storage-class specifier register (and auto) cannot appear in the declaration specifiers in an external declaration:

```
register extern int * p; // constraint-violation
```

The constraint that a register variable cannot have its address taken applies to all parts of the object. For example, if v is an array register variable then you cannot use any of the following in an expression:

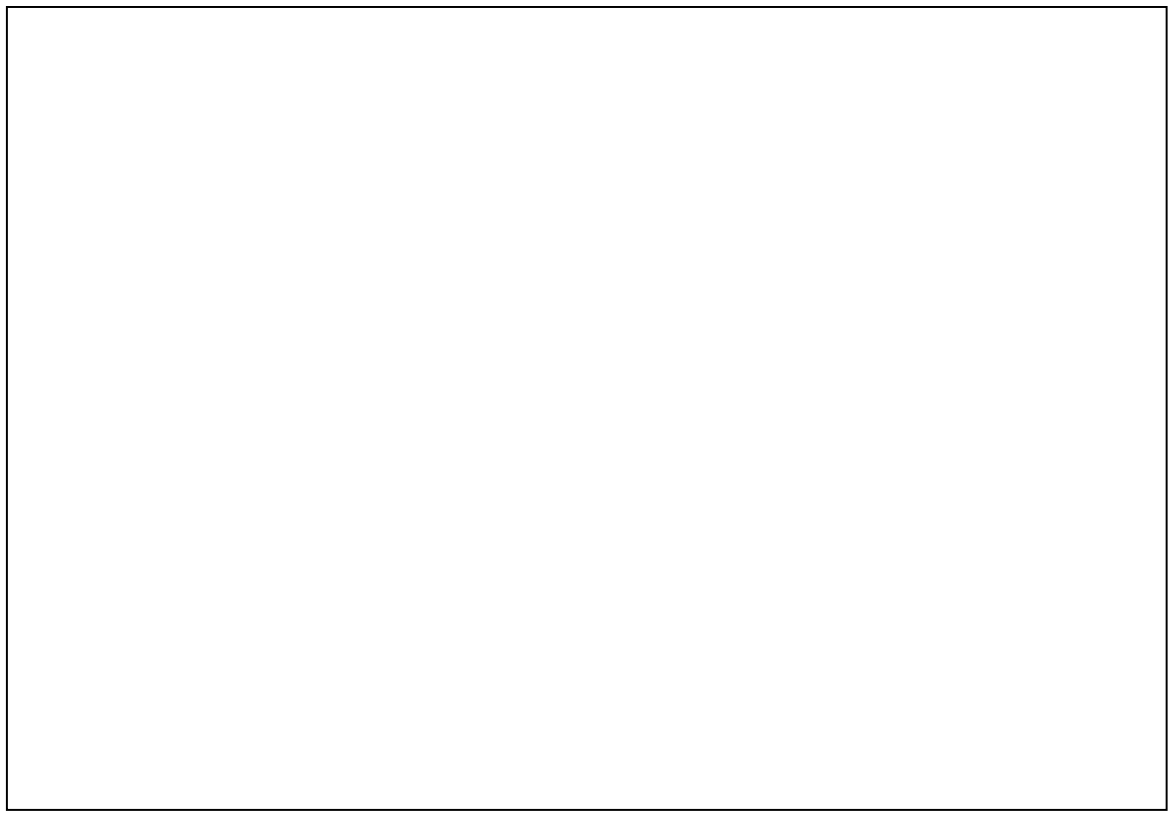
- v - since the name of an array decays into a pointer to its initial element.
- v[n] – since this is just a shorthand for *(v + n) and this specified v again

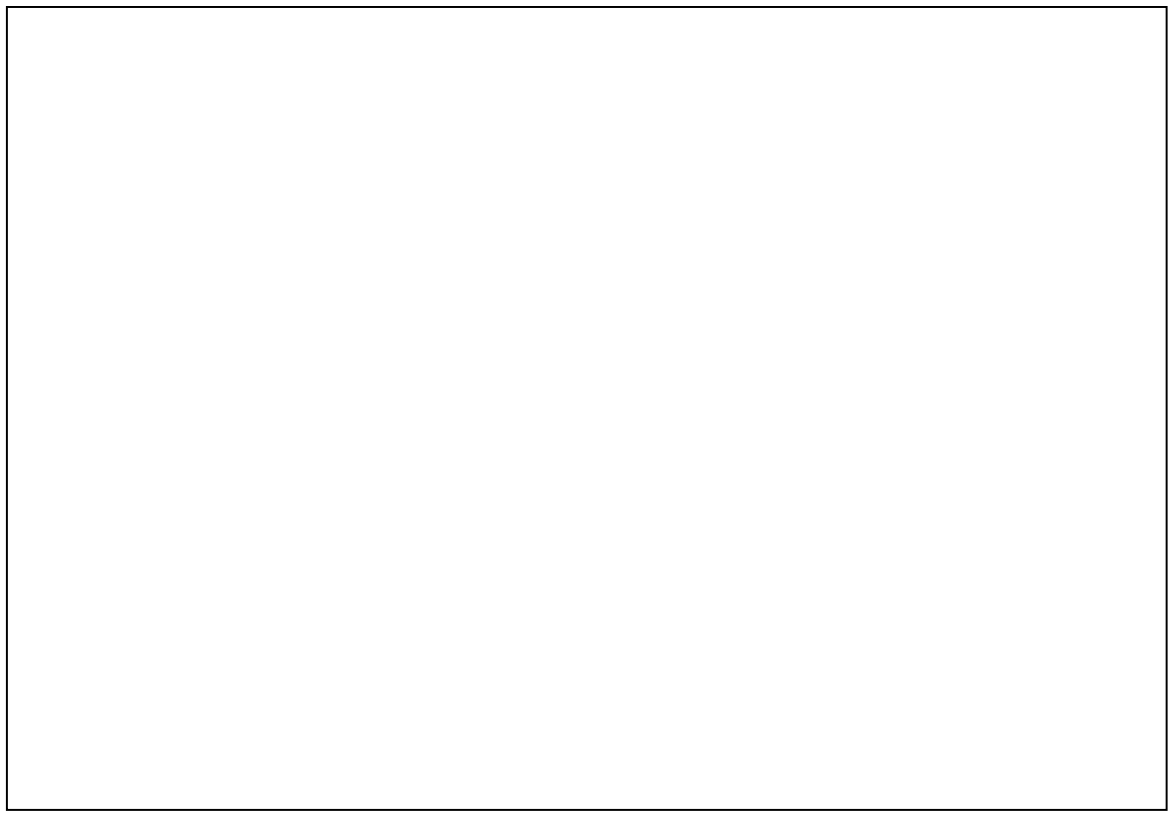
In fact the only operator that could be applied to a register array variable is sizeof.

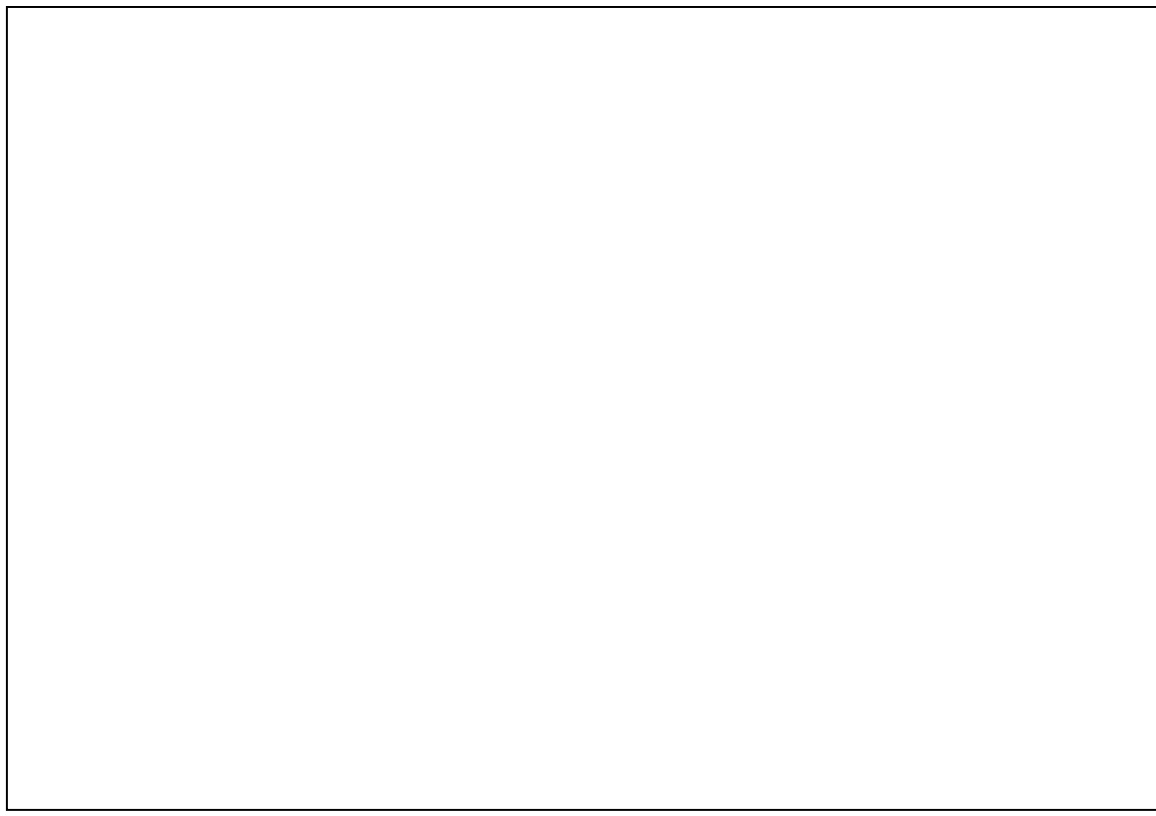


The remembers function returns the number of times it has been called!

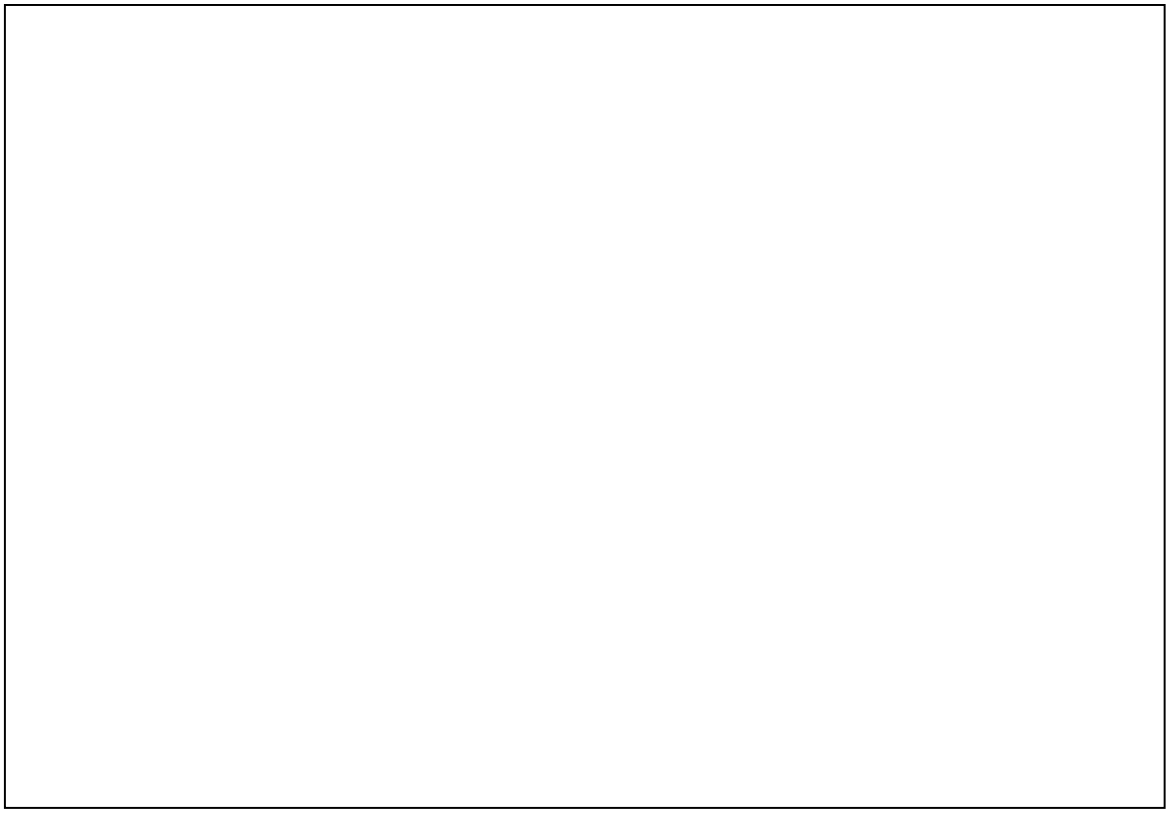
Beware, there are significant subtleties and re-entrancy issues with functions that retain state.







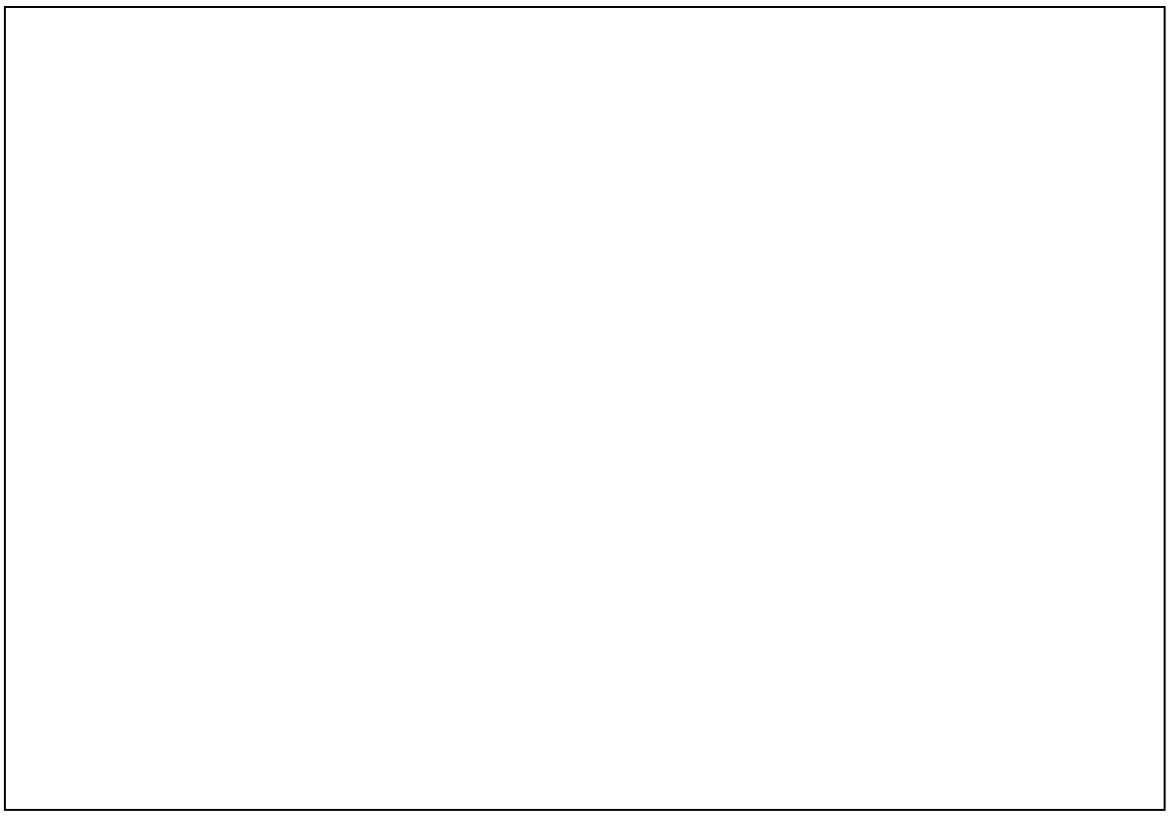
The linkage of an identifier is not part of its type. For example, a function pointer with external linkage can point to a function with internal linkage.



If a translation unit contains one or more tentative definitions for an identifier, and the translation unit contains no external definition for that identifier, then the behaviour is exactly as if the translation unit contains a file scope declaration of that identifier with an initializer equal to zero.

A declaration with the extern keyword and an initializer becomes a definition:

```
extern int v = 42;
extern int v = 42; // compile-time error duplicate-definition
```

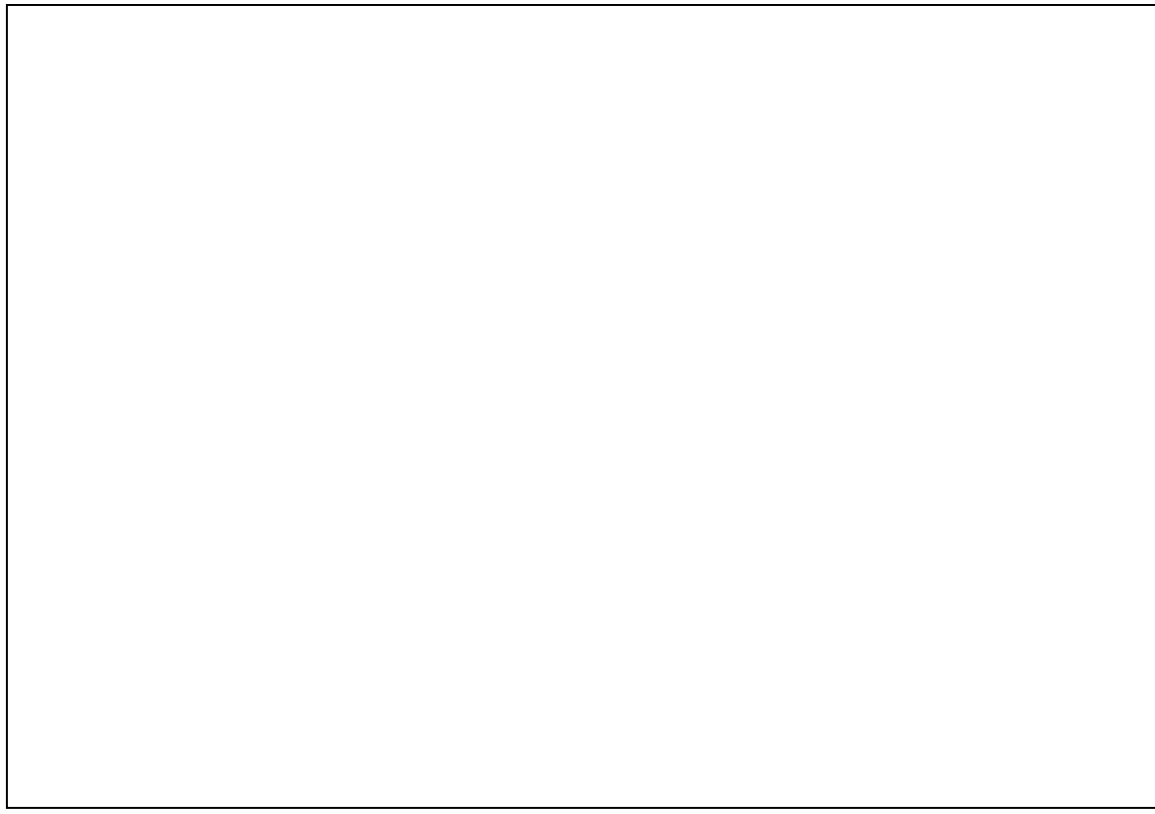


As well as internal and external linkage some identifiers have no linkage:

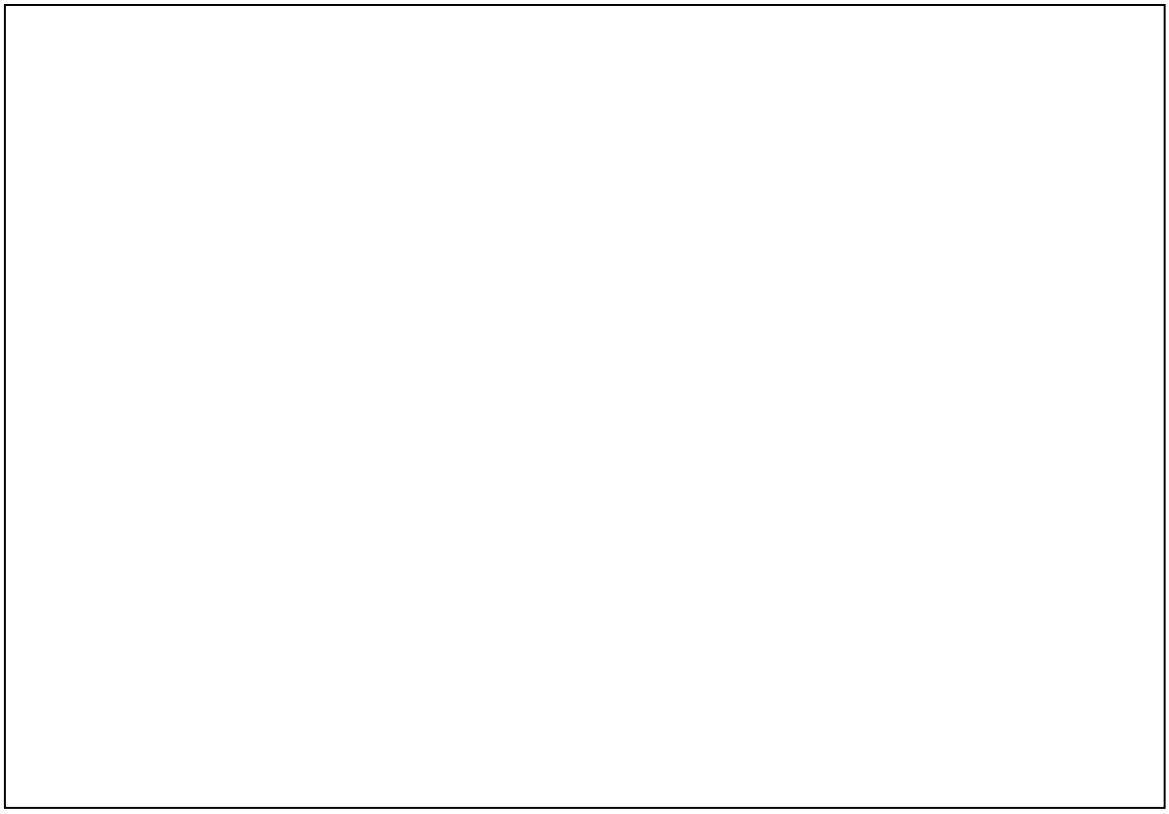
- an identifier declared to be anything other than an object or a function
- an identifier declared to be a function parameter
- a block-scope identifier for an object declared without the storage-class specifier extern

Note that C++ (but not C) forbids an identifier with no linkage being used to declare an identifier with linkage. For example:

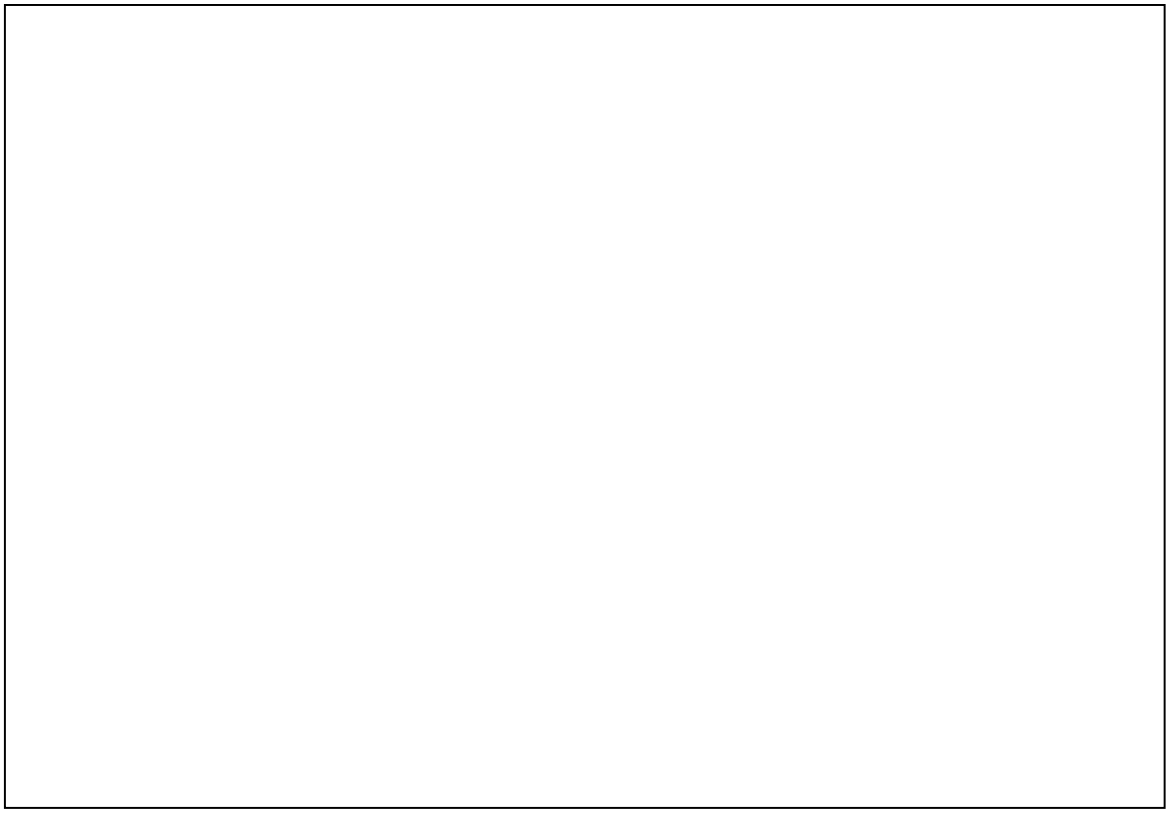
```
void eg(void)
{
   struct example { int i; };  // example has no linkage
   extern example a;     // conforming in C, ill-formed in C++
}
```



Inline functions were introduced in C99. They are borrowed from C, but are subtly different. Perhaps the simplest and most compatible approach is to define inline functions as static.



The assert() macros in <assert.h> prints the name of the file (from __FILE__), the line number (from __LINE__) and the enclosing function (from __func__) if the assertion fails.



In C (but not in C++) variadic functions must have at least one named parameter:
void illegal(…);

You need to be very careful about expression types when calling variadic functions. For example, the literal 0 (zero) is of type int. If you want to print the null pointer using the %p printf format specifier you must not write a plain zero since this will be of type int.

```
printf("%p", 0); // don't do this
```

Note that using the macro NULL is not safe either since NULL could be a macro for a plain zero:

```
printf("%p", NULL); // don't do this either
```

You must make sure the argument is a pointer:

```
printf("%p", (void*)0); // do this
```

Note also that expressions of a type smaller than int are automatically promoted to int when they bind to an ellipsis:

```
printf("%c", 'X'); // print a char
```

In this example 'X' will be promoted to an int – this means when the char is extracted using va_arg is must be extracted as an int and then cast to a char!