Functions and Program Structure

f'n 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>

struct FILE
{
    ...
};

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

return statement

- the *return* statement returns a value!
  - the expression must match the return type
  - either exactly or via implicit conversion
  - to end a *void* function early use *return*;

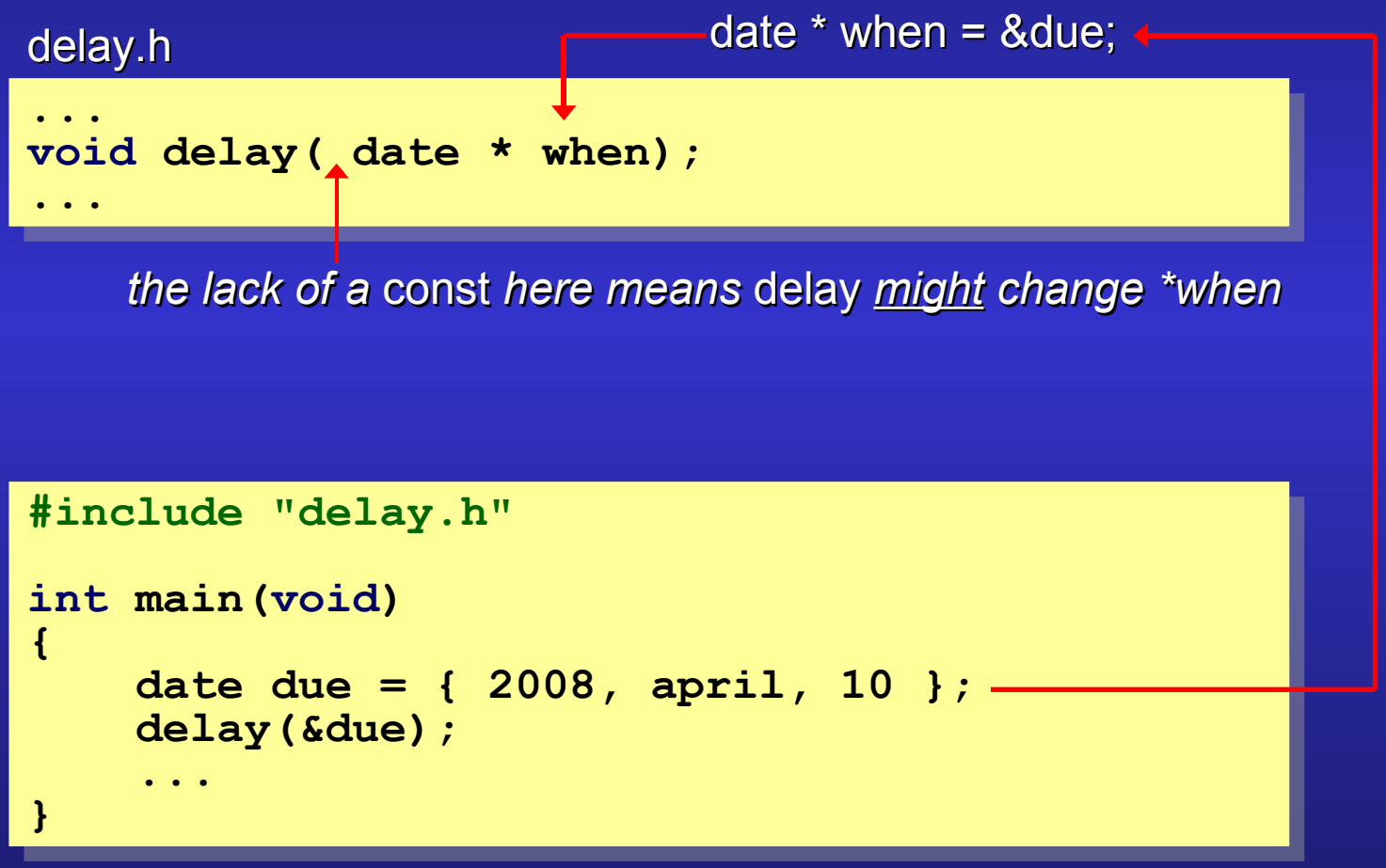
```
return expression_opt ;
```

```
const char * day_suffix(int days)
{
    const char * result = "th";

    if (days / 10 != 1)
        switch (days % 10)
        {
            ...
        }
    return result;
}
```

pass by pointer

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



pass by value

- 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);
    ...
}
```

pass by value

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

date.h

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

date when = today;

```
#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

pass by ptr to const

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

date.h

const date * when = &today;

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

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

```
#include "date.h"
...
int main(void)
{
    date today = { 2008, april, 10 };
    ...
    puts(day_name(&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
  - but make sure they are both small and stable

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)
{
    ...
}
```

auto variables

- the default storage class for local variables
  - life starts on entry to the block
  - life ends on exit from the block
  - does ***not*** have a default initial value
  - use if indeterminate causes ***undefined*** behaviour

```
size_t count(const int values[], size_t size, int find)
{
    size_t total = 0;
    ...
}
```

you write this

compiler sees as this (legal but massively non-idiomatic)

```
size_t count(const int values[], size_t size, int find)
{
    auto size_t total = 0;
    ...
}
```

register variables

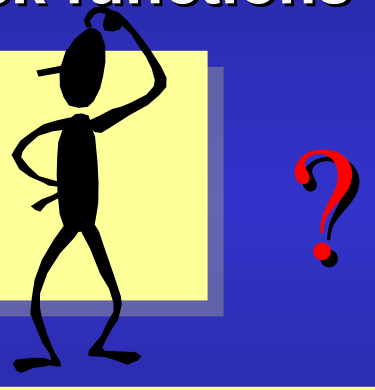
- a speed optimization hint to the compiler
  - compiler will use registers as best it can anyway
  - effect is implementation defined
  - register variables can't have their address taken

```
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);
    }
}
```

static local variables

- 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)
  - and often useful in unit test mock functions

```
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)
    {
        ...
    }
}
```

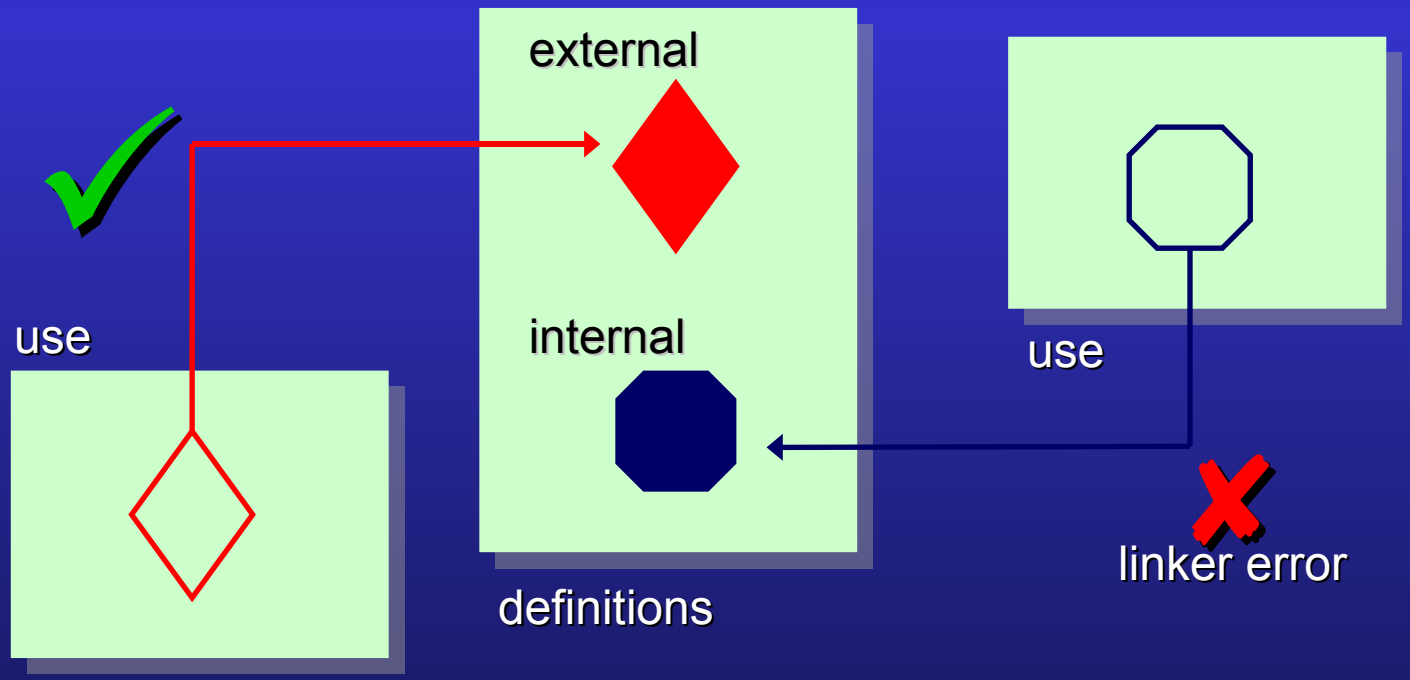
- a function can call itself
  - directly or indirectly
  - can often be rewritten using a loop

```
int factorial(int n)
{
    if (n <= 1)
        return 1;
    else
        return n * factorial(n - 1);
}
```

this will cause integer overflow for quite small values of n

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 linkage

- 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 linkage

- function definitions
  - 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 ISO C, duplicate definition errors in C++

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

- recommendation: extern data declarations
  - be explicit, use 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;
```

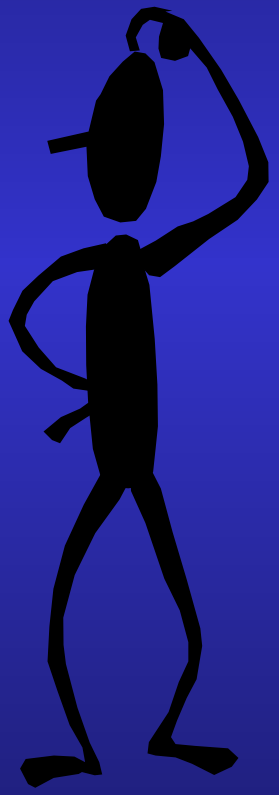
type linkage?

- 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 functions

- default to external linkage
  - creates a requirement for an "external" definition in another translation unit
  - simplest approach is to give all inline functions internal linkage
  - 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__);
}
```

as-if compiler translation

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

variadic functions

- 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(format, int)); break;
      case 'f': case 'F':
        print_double(va_arg(format, double)); break;
      ...
    }
  }
  va_end(args);
}
```

pointers 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 ptr args

- 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);
}
```

summary

- never define data in a header file
- mimic pass by copy using pointer to const for structs
- never use the auto keyword
- never 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

Contact

- This course was written by

Expertise: Agility, Process, OO, Patterns
Training+Designing+Consulting+Mentoring

{ JSL }

Jon Jagger

jon@ jaggersoft.com

www.jaggersoft.com