Advanced Techniques

- code that compiles in both C and C++
  - C++ compiler is stricter than C
  - e.g. pre C99 did not require function declarations
  - e.g. C++ requires more conversions to be explicit
  - avoid C++ keywords

bool	new	typeid
catch	operator	typename
class	private	using
const_cast	protected	virtual
delete	public	wchar_t
dynamic_cast	reinterpret_cast	
explicit	static_cast	
export	template	
false	this	
friend	throw	
mutable	true	
namespace	try	

C++ keywords that are not also C keywords

- prefer initialization to assignment

refactor

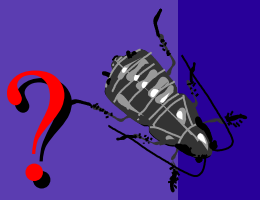
```
int count;
count = 0;
```

```
int count = 0;
```

- don't explicitly compare against true/false

refactor

```
if (has_prefix("is") == true)
    ...
```

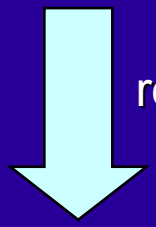


```
if (has_prefix("is"))
    ...
```

avoid redundant use of true/false

this version is very "solution focused"

```
bool is_even(int value)
{
    if (value % 2 == 0)
        return true;
    else
        return false;
}
```



refactor

this version is less solution focused;
it is more problem focused;
it is more "declarative"

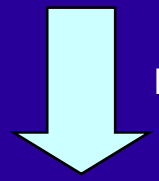
```
bool is_even(int value)
{
    return value % 2 == 0;
}
```

clarity ~ brevity

make inter-statement dependencies explicit

consider if you accidentally refactor the if without the last return - oops

```
bool somefunc(int value)
{
    if (value % 2 == 0)
        return alpha();
    return beta();
}
```

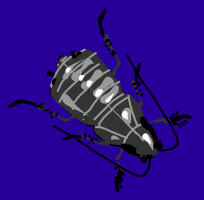


refactor

the return statements are now at the same indentation.
logical == physical

```
bool somefunc(int value)
{
    if (value % 2 == 0)
        return alpha();
    else
        return beta();
}
```

how to iterate



- iteration is a common bug hotspot
  - looping is easy, knowing when to stop is tricky!
- follow the fundamental rule of design
  - always design a thing by considering it in its next largest context
  - what do you want to be true *after* the iteration?
  - this forms the termination condition

```
bool search(int values[], size_t end, int find)
{
    size_t at = 0;
    // values[at==0 → at==end] iteration
    at == end || values[at] == find
    ...
}
```

we didn't find it

the short-circuit here is vital

we found it

- the negation of the termination condition is the iteration's continuation condition
  - !(at == end || values[at] == find)
  - at != end && values[at] != find

equivalent
(DeMorgan's Law)

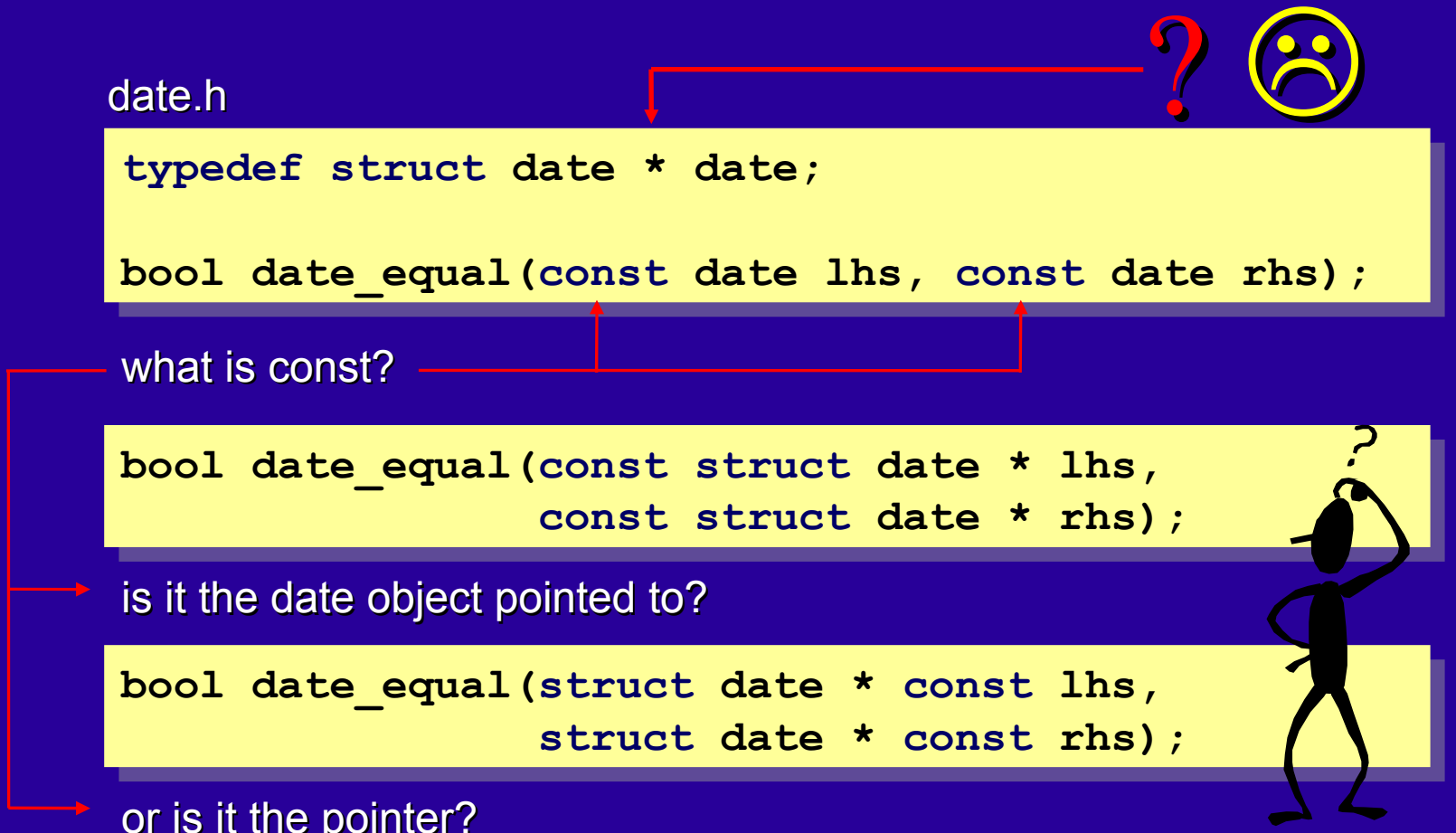
```
bool search(int values[], size_t end, int find)
{
    size_t at = 0;

    while (at != end && values[at] != find)
    {
        ...
    }
    ...
}
```

then simply fill in the loop body

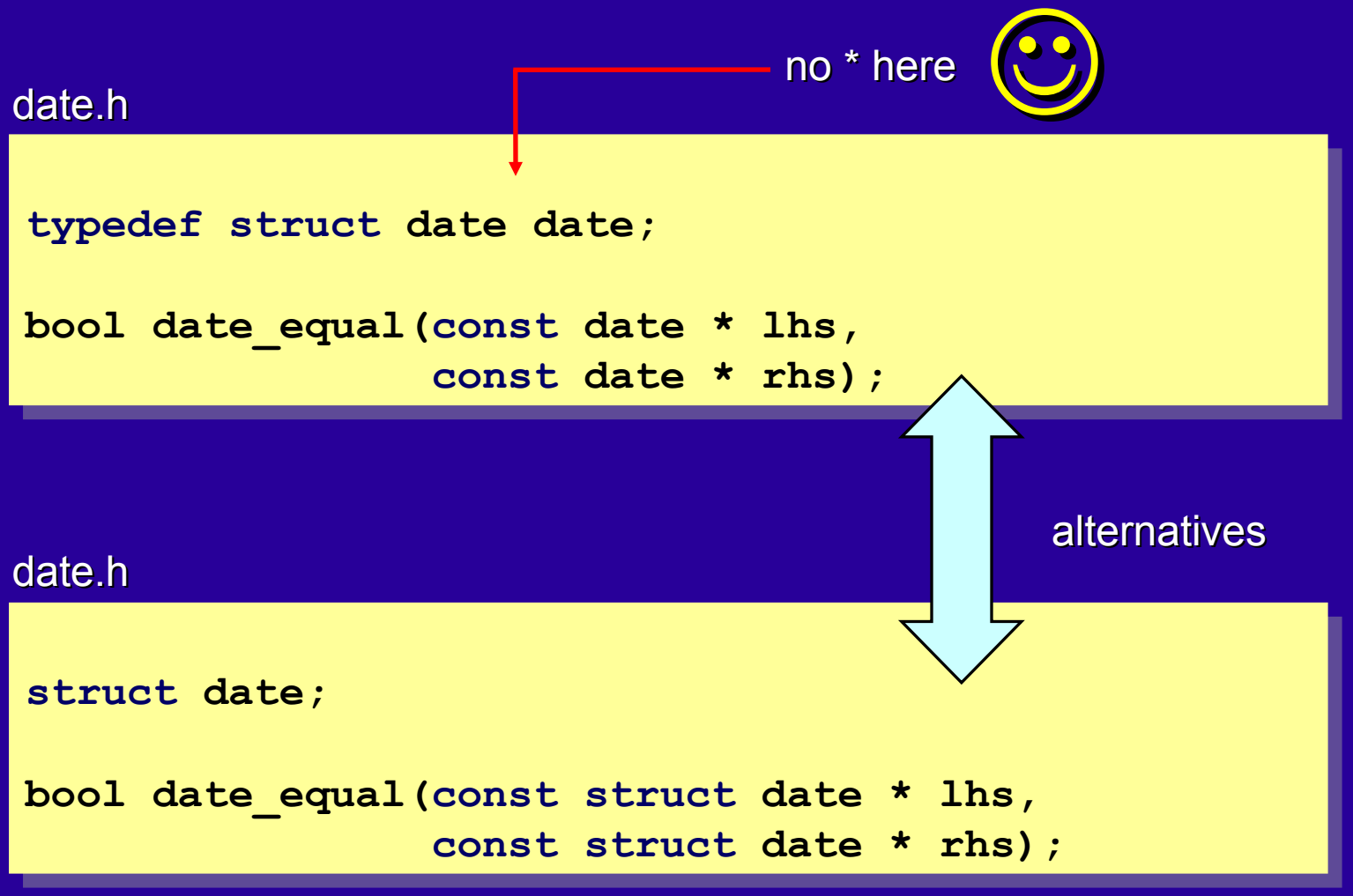
```
    at++;
```

bad typedef

- don't attempt to hide a pointer in a typedef
  - if it's a pointer make it look like a pointer
  - abstraction is about hiding *unimportant* details



good typedef



- a typedef does *not* create a new type

```
typedef int mile;
typedef int kilometer;
```

```
void weak(mile lhs, kilometer rhs)
{
    lhs = rhs;
}
```

- consider using a wrapper type instead…

```
struct mile { int value; };
struct kilometer { int value; };
```

```
void strong(mile lhs, kilometer rhs)
{
    lhs = rhs;
}
```

- enums are very weakly typed
  - an enum's enumerators are of type integer, not of the enum type itself!

```
enum suit
{
    clubs, diamonds, hearts, spades
};
```

```
enum season
{
    spring, summer, autumn, winter
};
```

```
void weak(void)
{
    suit trumps = winter;
}
```

an enum can also be wrapped in a struct

```
struct suit
{
    enum
    {
        clubs, diamonds, hearts, spades
    } value;
};
```

```
struct season
{
    enum
    {
        spring, summer, autumn, winter
    } value;
};
```

- 5.1.2.3 Program semantics
  - At certain specified points in the execution sequence called sequence points,
    - all *side effects* of previous evaluations shall be complete and
    - no *side effects* of subsequent evaluations shall have taken place
- what constitutes a side effect?
  - accessing a volatile object
  - modifying an object
  - modifying a file
  - calling a function that does any of these

volatile

6.7.3 Type qualifiers

- An object that has volatile-qualified type may be modified in ways unknown to the implementation or have other unknown side effects. Therefore any expression referring to such an object shall be evaluated strictly according to the rules…described in 5.1.2.3

```
int global;
volatile int reg;
...
    reg *= 1;

    reg = global;
    reg = global;

    int v1 = reg;
    int v2 = reg;
...
```

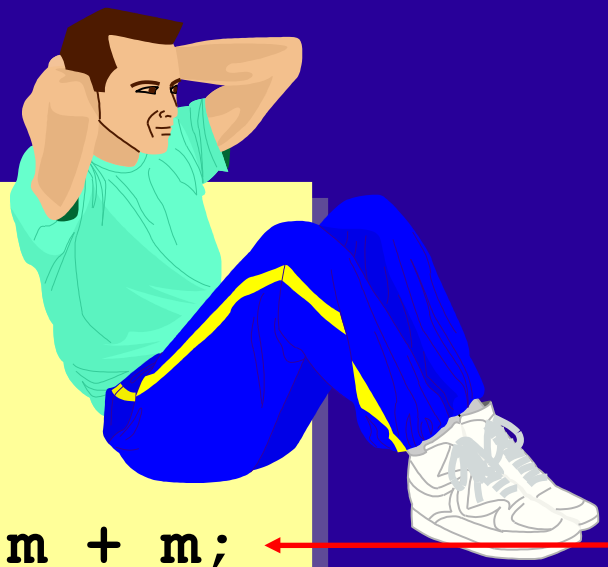
reg looks unchanged but reg is volatile so an access to the object is required. This access may cause its value to change.

these cannot be optimized to a single assignment.

v1 might not equal v2.

exercise

- in this statement…
  - where are the sequence points?
  - where are the side-effects?
  - is it undefined?

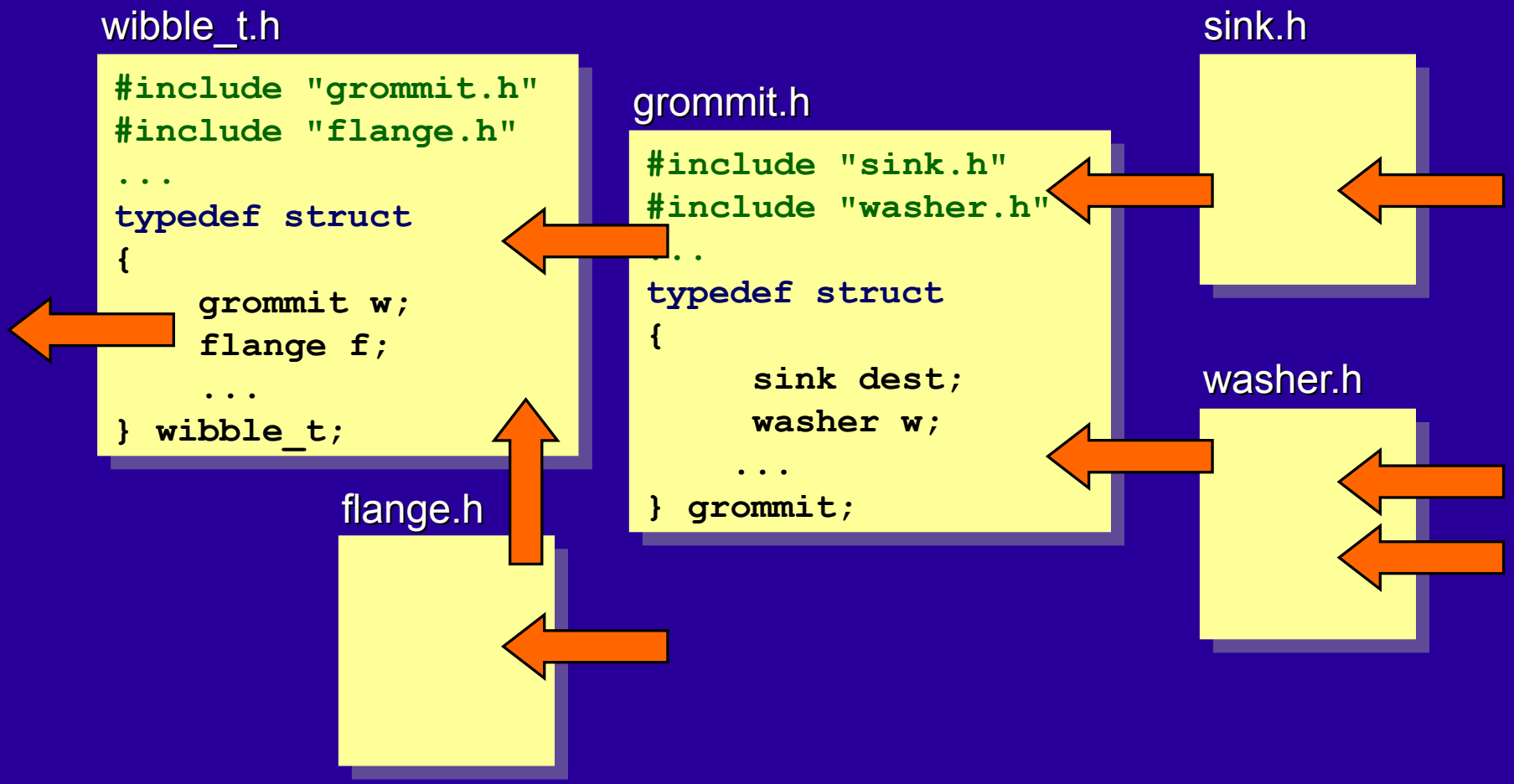


```
volatile int m;

void eg(void)
{
    int value = m + m;
    ...
}
```

dependencies

#include dependencies are *transitive*

- if you change a .h file you have to recompile all files that #include it at any depth
- a visible reflection of the physical coupling



- some #includes in headers are not necessary

eg.h

```
//#include "wibble.h"
struct wibble;

struct wibble * eg1(void);
void eg2(struct wibble *);

struct wibble   eg3(void);
void eg4(struct wibble  );
```

when eg3 and eg4 are *called in an expression* the compiler will need to know the size of a wibble – but these are not expressions they are declarations.

- an ADT implementation technique
  - a forward declaration gives the name of a type
  - the definition of the type – and it's accompanying #includes – are *not* specified in the header
  - all use of the type has to be as a pointer and all use of the pointer variable has to be via a function

wibble.h

```
...

struct wibble;

struct wibble * wopen(void);

int wclose(struct wibble * stream);
...
```

minimal #includes

forward declaration

all uses of wibble have to be as pointers

- in most APIs the idea that a set of functions are closely related is quite weakly expressed

```
int main(int argc, char * argv[])
{
    wibble * w = wopen(argv[1]);
    ...
    wclose(w);
}
```

- a struct containing function pointers can express the idea more strongly

```
int main(int argc, char * argv[])
{
    wibble * w = wibbles.open(argv[1]);
    ...
    wibbles.close(w);
}
```

module header

wibble.h

```
#ifndef WIBBLE_INCLUDED
#define WIBBLE_INCLUDED
...
...
struct wibble;

struct wibble_api
{
    struct wibble * (*open)(const char *);
    ...
    int (*close)(struct wibble *);
};
extern const struct wibble_api wibbles;

#endif
```

module source

wibble.c

```
#include "wibble.h"
...
...
static
wibble * open(const char * name)
{     ...
}
...
static
int close(wibble * stream)
{     ...
};
const struct wibble_api wibbles =
{
     open, ..., close
};
```

static linkage

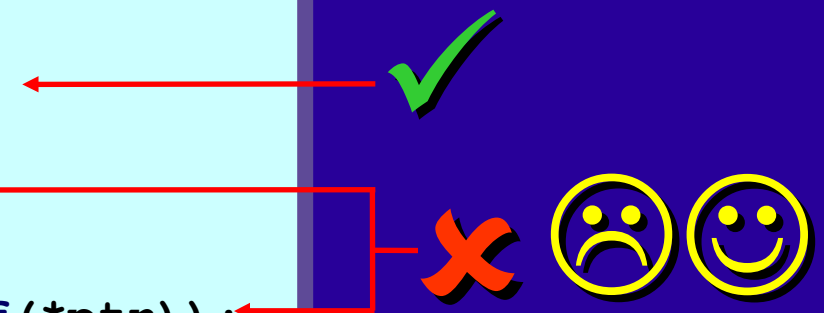
no need to write
&open, &close

- opaque type memory management…
  - clients cannot create objects since they don't know how many bytes they occupy

wibble.h

```
...
struct wibble;
...
```

```
#include "wibble.h"

int main(void)
{
    wibble * pointer;
    ...
    wibble value;
    ...
    ptr = malloc(sizeof(*ptr));
}
```



- an ADT can declare its size!
  - clients can now allocate the memory
  - everything else remains abstract

wibble.h

```
struct wibble
{
    unsigned char shadow[16];
};

bool wopen(struct wbble *, const char *);
```

```
#include "wibble.h"
int user(int argc, char * argv[])
{
   const char * name = argv[1];
   wibble w;
   if (wopen(&w, name))
        ...
}
```

- implementation needs to…
- define the true type being shadowed

wibble.h

```
struct wibble
{
    unsigned char shadow[16];
};
```

wibble.c

```
#include "grommit.h"
#include "flange.h"

struct shadowed_wibble
{
    grommit g;
    flange f;
    ...
};
```

the analogy is that the shadowed type casts a shadow which reveals only its size

- implementation needs to…
- convert between the two types

```
bool wopen(wibble * w, const char * name)
{
    shadowed wibble * s = (shadowed_wibble *)w;
    s->grommit = ...;
    s->flange = ...;
    ...
}
```

implementation can simply do a cast…

wibble.c

```
bool wopen(wibble * w, const char * name)
{
    shadowed_wibble s;
    memcpy(&s, w, sizeof(s));
    s.grommit = ...;
    s.flange = ...;
    memcpy(w, &s, sizeof(s));
    ...
}
```

…or it can do a memcpy

alignment?

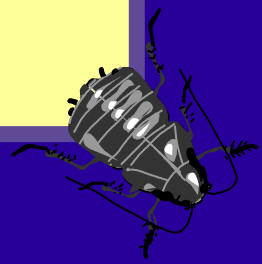
- the two types must have the same alignment
  - this means the first member of both types must have the same alignment

wibble.h

```
struct wibble
{
    unsigned char shadow[16];
};
```

wibble.c

```
struct shadowed_wibble
{
    grommit g;
    flange f;
    ...
};
```



- create a union of all the basic types!
- the union's alignment will reflect the maximal alignment

aligned.h

```
union aligned
{
    char c,*cp;
    short s, * sp;
    int i,*ip; long l,*lp; long long ll,*llp;
    float f,*fp; double d,*dp; long double ld,*ldp;
    void *vp;
    void (*fv)(void); void (*fo)(); void (*fe)(int,...);
    struct s * ssp;
    union u * uup;
};
```

- create another union; an array of bytes for the size plus alignment

wibble.h

```
#include "aligned.h"
...
union wibble
{
    union aligned correctly;
    unsigned char size[16];
};
...
```

this for alignment

this for size

- unions are not-common...
- so wrap the union inside a struct

wibble.h

```
#include "aligned.h"
...
struct wibble
{
    union
    {
        union aligned correctly;
        unsigned char size[16];
    } shadow;
};
...
```

size problem

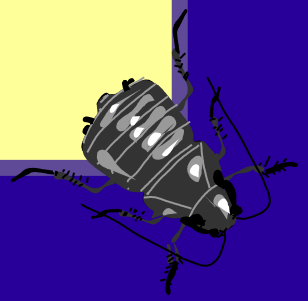
- the size of the type must not be smaller than the size of the type it shadows
  - assert only works at runtime
  - it would be safer to check at compile time

wibble.h

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

wibble.c

```
struct shadowed_wibble
{
    ...
};
```



```
assert(sizeof(wibble) >= sizeof(shadowed_wibble))
```

- use a compile time assertion
  - you cannot declare an array with negative size

cta.h

```
#define COMPILE_TIME_ASSERTION(assertion) \¬
    extern char CTA_NAME [ (assertion) ? 1 : -1 ]
#define CTA_NAME \¬
    CTA_CAT(compile_time_assertion_at_line_, __LINE__)
#define CTA_CAT(lhs,rhs)        CTA_CAT_AGAIN(lhs,rhs)
#define CTA_CAT_AGAIN(lhs,rhs)  lhs ## rhs
```

```
#include "cta.h"
COMPILE_TIME_ASSERTION(1 == 1);
```

```
#include "cta.h"
COMPILE_TIME_ASSERTION(1 == 0);
```

```
gcc→error: size of array
'compile_time_assertion_at_line_2' is negative
```

size solution

wibble.c

```
#include "wibble.h"
#include "cta.h"
#include "flange.h"
#include "grommet.h"

struct shadowed_wibble
{
    grommet g;
    flange f;
} shadowed_wibble;

COMPILE_TIME_ASSERTION(
    sizeof(wibble) >= sizeof(shadowed_wibble));
```

spirit of C

- trust the programmer
  - let them do what needs to be done
  - the programmer is in charge not the compiler
- keep the language small and simple
  - small amount of code → small amount of assembler
  - provide only one way to do an operation
  - new inventions are not entertained
- make it fast, even if its not portable
  - target efficient code generation
  - int preference, int promotion rules
  - sequence points, maximum leeway to compiler
- rich expressions
  - lots of operators
  - expressions combine into larger expressions

books

chapter and verse!

- The C Programming Language
  - Kernighan and Ritchie
- The C Standard
  - BSI
- C: A Reference Manual
  - Harbison and Steele
- C FAQ's
  - Steve Summit
- Expert C Programming
  - Peter van der Linden
- Safer C
  - Les Hatton

Note that the bool, false, and true keywords need careful attention if your code has to compile in both C and C++.

A C++ compiler will #define the preprocessor token __cplusplus whereas a C compiler will not. You can use this to conditionally include/exclude code depending on what language the source file is being compiled in:

```
#if defined(__cplusplus)
 … this will be seen only if compiling in C++
#else
 … this will be seen if not compiling in C++ (assumed to be C)
#endif
```

Explicitly comparing against true/false is not only bad style it is also dangerous. This is because being equal to true is not the same as being unequal to false. Remember, in C, *any* non-zero value will be interpreted as false but the keyword false equals the specific value one.

The simplest way to look at this code is that the first fragment spells out, in tedious detail, what is involved in arriving at either true or false — "if value is perfectly divisible by two, then the result is true, otherwise it is false". In the second slide we simply "return whether or not value is perfectly divisible by 2".

Put similar concepts at a similar level. The example above could be further reduced by using the conditional operator, if the use of alpha and beta lends itself naturally to that kind of compaction.

A common variation of the above is to return the position of the element if found or one-beyond-the-end if not found:

```
int * search(int values[], size_t end, int find)
{
    size_t at = 0;
    while (at != end && values[at] != find) {
        at++;
    }
    return &values[at];
}
```

The slide asks the question: what is const? is it the date object pointed to? or is it the pointer itself?

Answer: it is the pointer.

An important factor when deciding which of the above styles to adopt might well be which is the dominant current style. If the codebase currently favours the former style you should lean towards the former style, whereas if the codebase currently favours the later style you should favour the later style. Consistency is important.

When all other factors are equal, I (Jon Jagger) personally prefer the former style (and was recently encouraged to see it is also a style Brian Kernighan adopts; Brian Kernighan is the K in K&R).

Note that typedef'ing a struct tagname to the same name causes an error in C++. If you need C++ compatibility you can choose a different name for the tagname:

```
typedef struct date_tag date;
```

or make the typedef conditional on not compiling in C++

```
struct date;
#if !defined(__cplusplus__)
typedef struct date date;
#endif
```

Note that not every side-effect or evaluation can be authoritatively determined to be either previous to or subsequent to a given sequence point. For example, given

```
x = 0;
b = (x = 1, 2 * x) + 3 * x;
```

The evaluation of 3*x is not ordered with respect to the sequence point introduced by the comma. Note that x is not modified between sequence points so (as long as x is not volatile) the constraint introduced in clause 6.5 " Between the previous and next sequence point an object *shall* have its stored value modified at most once by the evaluation of an expression." is not violated.

The clear message is that in C the onus is on the programmer to strive to make their expressions and statements as simple as possible.

The three type qualifiers are const, restrict, and volatile.

If the specification of an array type includes any type qualifiers, the element type is so qualified not the array type. In other words the following declares 42 ints all of which are volatile:

```
volatile int array[42];
```

Note that in a multi-threaded environment the volatile keyword does *not* provide any atomicity or synchronization guarantees.

The specified statement does not contain any embedded sequence points.

Merely accessing a volatile is a side-effect so the statement contains two side effects.

Yes, it is undefined, since m is modified twice between sequence points.

In UML dependencies are drawn as an arrow pointed towards the dependent item. On the slide the arrows are drawn in the opposite direction to emphasize the dynamic nature of the ripple of change.

The most distant transitive dependencies for a module is sometimes known as its dependency horizon. Unit testing is a very good way to explicitly expose dependency horizons.

Note that the first line can be split into two:

```
struct wibble_tag;
typedef wibble_tag wibble;
```

Note that the function pointers inside the struct have to be declared explicitly (*open). They cannot be declared like this:

```
struct wibble_api
{
   wibble * open(const char *); // compile time error
   …
};
```

The rule that the name of a function decays into a pointer to the function occurs only in expressions.

An interesting and useful property of using the memcpy overlay technique is that it makes the operation atomic from the user's perspective. If the implementation has modified the shadow value but not memcpy'd back to the user's memory then as far as the user is concerned nothing has happened.

Another approach to achieving alignment compatibility is to give both the user type and the shadow type a common first member. There are techniques that use this approach.

A compiler has to give a union an alignment no worse that the most strictly aligned member of the union. If we create a union with all the basic types we effectively create a type that we can use to force strict alignment. We simply combine (in another union!) the union type with the type we want aligned.

A compiler has to give a union an alignment no worse that the most strictly aligned member of the union. If we create a union with all the basic types we effectively create a type that we can use to force strict alignment. We simply combine (in another union!) the union type with the type we want aligned.

A compiler has to give a union an alignment no worse that the most strictly aligned member of the union. If we create a union with all the basic types we effectively create a type that we can use to force strict alignment. We simply combine (in another union!) the union type with the type we want aligned.

Note that we use the >= operator in assert(sizeof(wibble) >= sizeof(shadowed_wibble)) rather than the stricter == operator. This allows binary compatibility with an alternative, smaller representation.

This compile time assertion relies on the fact that Standard C does not allow you to define an array of negative length. (Standard C also forbids you from declaring an array of length zero, but some compilers are a bit lax on enforcing this constraint).

The repeated concatenation is required because the ## operator does not perform any macro expansion of its arguments. That is, when processing lhs##rhs if lhs or rhs is __LINE__ then the concatenation will cause the symbol __LINE__ to literally become part of the new symbol rather than the value of __LINE__ becoming part of the new symbol.

Notice how the name of the array is formed so the error message from gcc is very intention revealing. It even finishes the message "is negative" which can be read in the logical sense rather than the arithmetic sense.

Also worth knowing about is a free PDF book on the C Standard written by Derek Jones, one of the world's foremost experts on C:

http://www.knosof.co.uk/cbook/cbook.html

This book contains a commentary on every single sentence in the C Standard. It is invaluable if you are reading the C Standard and need some help understand what specific sentence means.

Also recommended is Andrew Koenig's C Traps and Pitfalls.