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

clarity ~ brevity

- prefer initialization to assignment

refactor

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

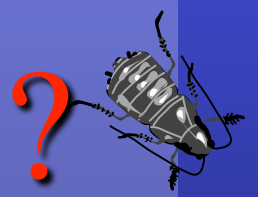
```
int count = 0;
```

- don't explicitly compare against true/false

refactor

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

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

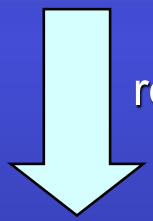


- **avoid redundant use of true/false**

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"

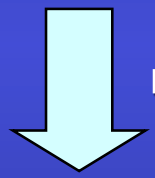
```
bool is_even(int value)
{
    return value % 2 == 0;
}
```

implicit vs explicit

- **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();
}
```

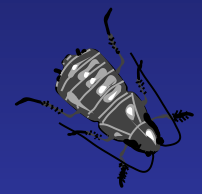


refactor

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

how to iterate

- 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)

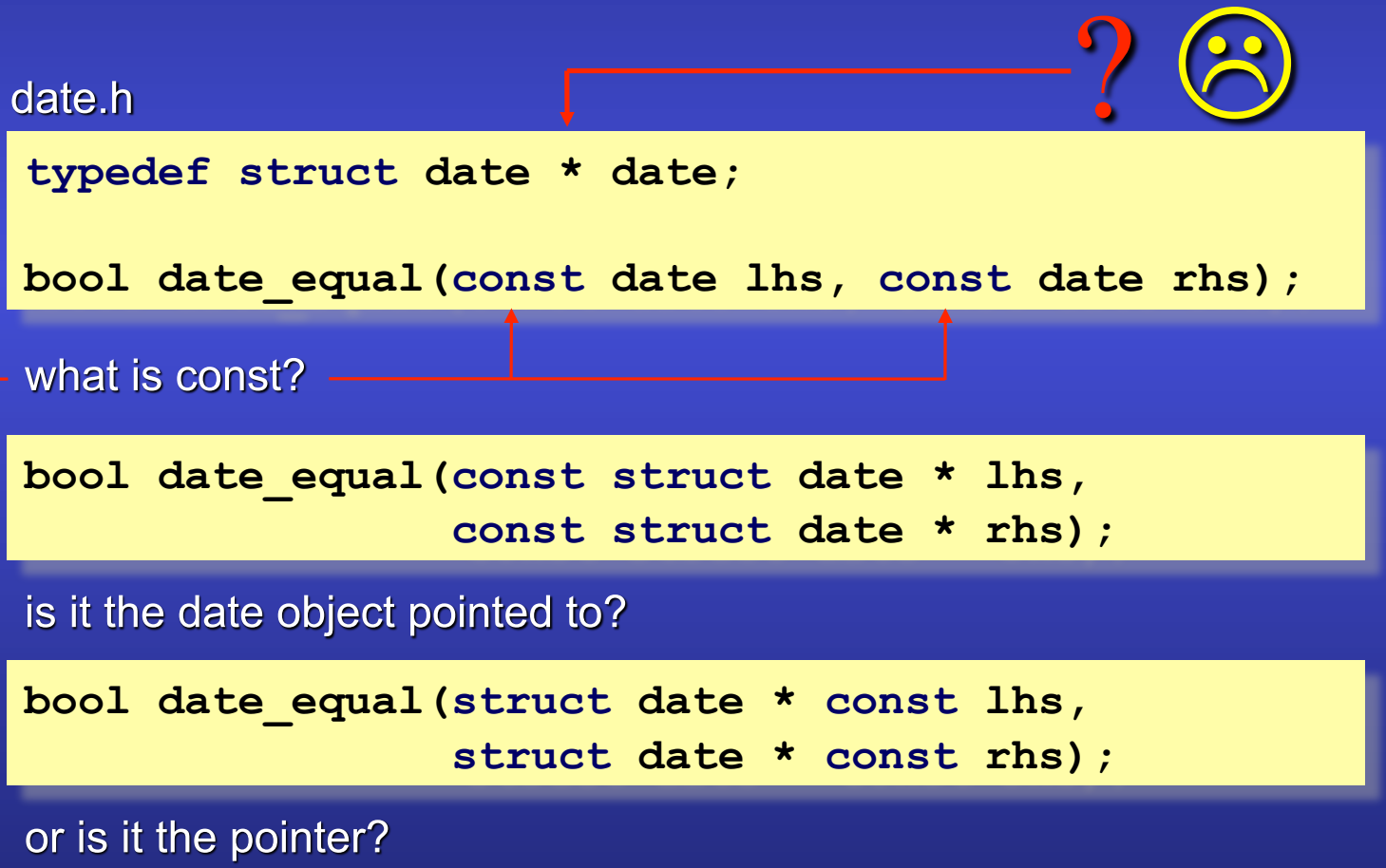
```
bool search(int values[], size_t end, int find)
{
    size_t at = 0;

    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

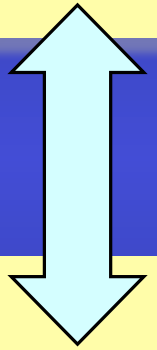


good typedef

no * here

date.h

```
typedef struct date date;

bool date_equal(const date * lhs,
                const date * rhs);
```



alternatives

date.h

```
struct date;

bool date_equal(const struct date * lhs,
                const struct date * rhs);
```

strong 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;
}
```

weak enum

- 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;
}
```

stronger enums

- 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;
};
```

side effects

- 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;
...
```

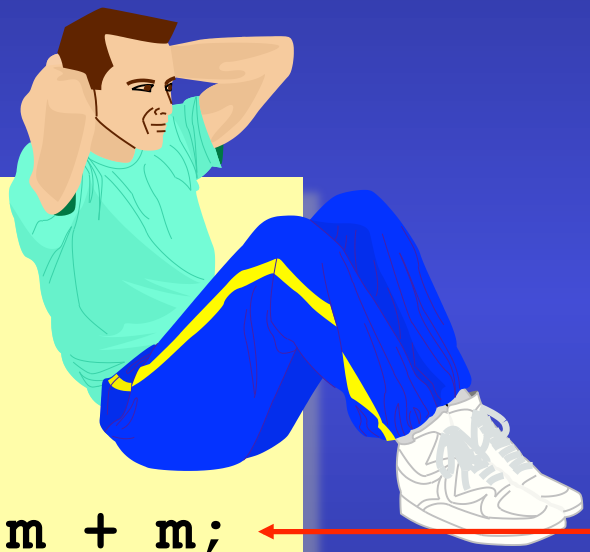
reg looks unchanged but reg is volatile so an access to the object is required. This access may cause its value to change.

these cannot be optimized to a single assignment.

v1 might not equal v2.

exercise

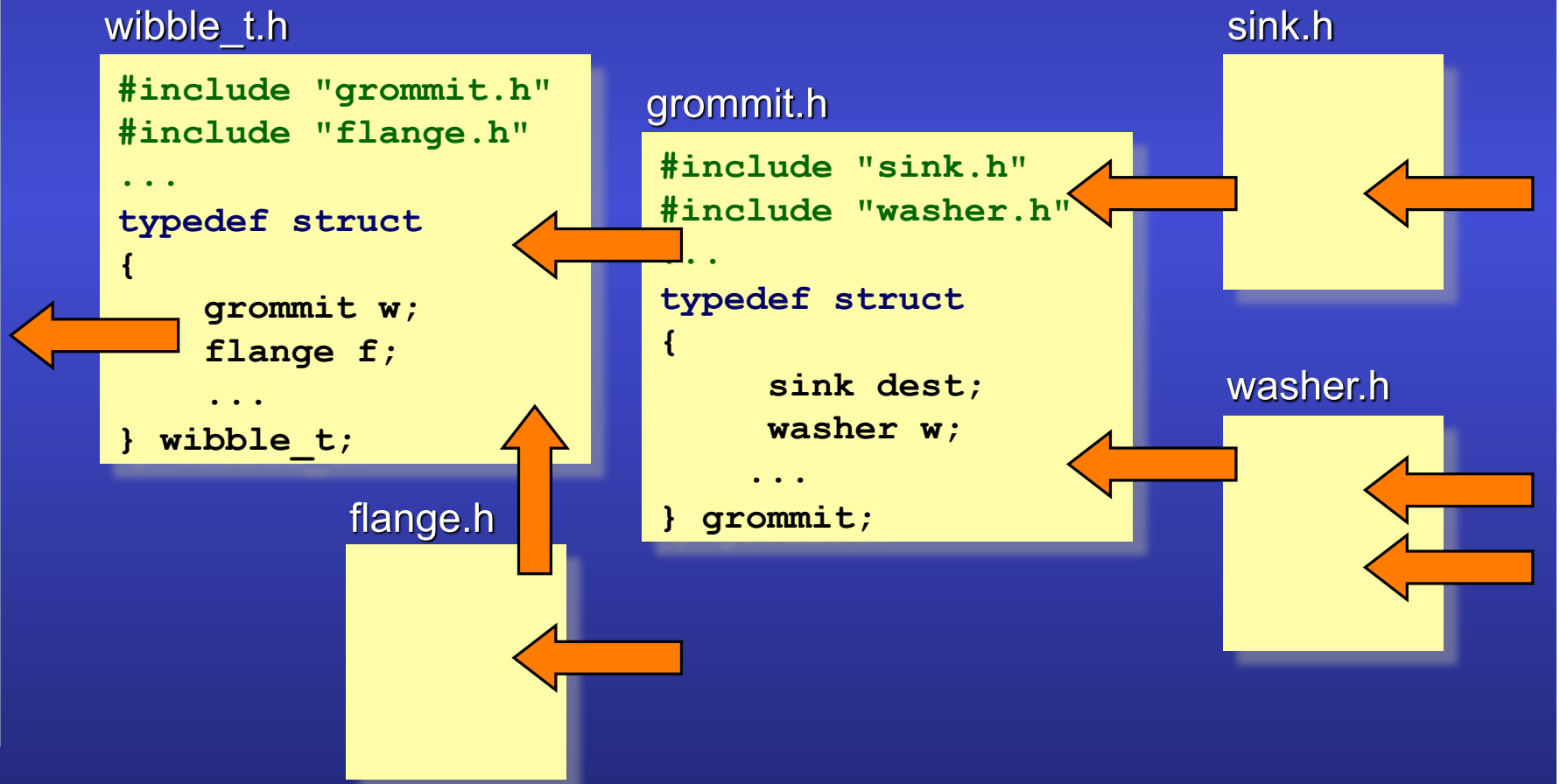
- in this statement…
  - where are the sequence points?
  - where are the side-effects?
  - is it undefined?



```
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



opaque types

- 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

module : use

- 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));
}
```

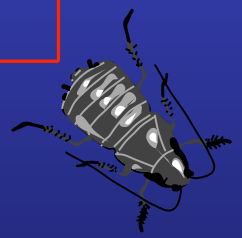
shadow data type

- 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 wibble *,
           const char *);
```

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



shadowed type

- 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 problem

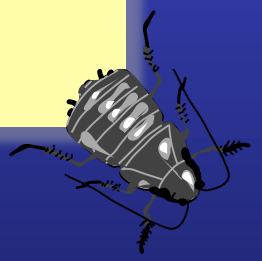
- the two types must have the same alignment
  - this means the first member of both types must have the same alignment

wibble.h

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

wibble.c

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



alignment solution

- 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;
};
```

alignment solution

- 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

alignment solution

- 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

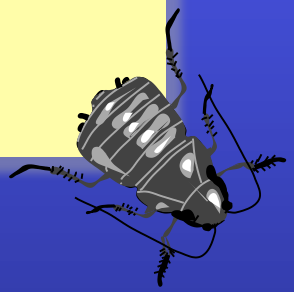
- the size of the type must not be smaller than the size of the type it shadows
  - assert only works at runtime
  - it would be safer to check at compile time

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));
```

books

- 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

chapter and verse!

testability

- functions using internally wired file scope data are hard to test
  - the Parameterize from Above (PfA) pattern can help to loosen the tight coupling

```
void not_easy_to_test(void)
{
    printf(...);
    printf(...);
    ...
}
```

?

```
void easier_to_test(FILE * stream)
{
    fprintf(stream, ...);
    fprintf(stream, ...);
    ...
}
```

✓

```
void not_easy_to_test(void)
{
    easier_to_test(stdout);
}
```

a global

testability

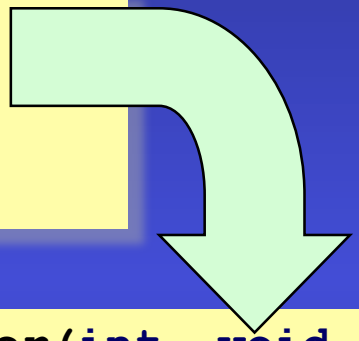
- better – but still not a ***unit*** test

```
size_t fsize(FILE * stream);
int freadall(FILE * stream, char buffer[], size_t n);

void example_test(void)
{
    FILE * stream = fopen("test.txt", "w");
    easier_to_test(stream);
    fclose(stream);

    stream = fopen("test.txt", "r");
    size_t n = fsize(stream);
    char * actual = malloc(n);
    freadall(stream, actual, n);
    fclose(stream);

    const char * expected = "42";
    bool same = strcmp(expected, actual) == 0;
    free(actual);
    assert(same);
}
```

testability

- a unit test cannot touch the file system
  - use PfA again; make the parameter a function ptr
  - note the external API is again unchanged

```
void not_easy_to_test(FILE * stream)
{
    int result = fputc('4', stream);
    ...
}
```



```
void easy_to_test(void * v, int putter(int, void *))
{
    int result = putter('4', v);
    ...
}
```

```
int fputc_stub(int ch, void * stream)
{
    return fputc(ch, (FILE*)stream);
}
```

```
void not_easy_to_test(FILE * stream)
{
    easy_to_test(stream, fputc_stub);
}
```

testability

- now a proper unit test
  - isolated, repeatable, automatable, fast

```
void easy_to_test(void * v, int putter(int, void *))
{
    int result = putter(v, ch);
    ...
}
```

```
int my_putter(int c, void * v)
{
   char * ptr = v;
   *ptr = c;
   return c;
}

void example_test(void)
{
    char put[] = { '\0' };
    easy_to_test(put, my_putter);
    assert(put[0] == '4');
}
```

testability

- testing
  - no one aspect of software development is more encompassing than testing
  - make unit testing compulsory
- dependencies
  - unit testing helps to ensure active management of dependencies
  - it helps to reveal the dependency horizon
  - Parameterize from Above
  - Don’t Talk To Strangers

TESTING TESTING