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

C++ keywords that are not also C keywords:

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

void test_is_leap_year(void)
{
    // years not divisible by 4 are leap years
    assert(is_leap_year(1906));
    assert(!is_leap_year(2009));

    // years divisible by 4 but not 100 are leap years
    assert(is_leap_year(1984));
    assert(is_leap_year(2008));

    // years divisible by 100 but not 400 are not leap years
    assert(!is_leap_year(1900));
    assert(!is_leap_year(2100));

    // years divisible by 400 are leap years
    assert(is_leap_year(2000));
    assert(is_leap_year(2400));
}
• Imagine if C had no comments

void years_not_divisible_by_4_are_leap_years(void)
{
    assert(is_leap_year(1906));
    assert(!is_leap_year(2009));
}

void years_divisible_by_4_but_not_100_are_leap_years(void)
{
    assert(is_leap_year(1984));
    assert(is_leap_year(2008));
}

void years_divisible_by_100_but_not_400_are_not_leap_years(void)
{
    assert(!is_leap_year(1900));
    assert(!is_leap_year(2100));
}

void years_divisible_by_400_are_leap_years(void)
{
    assert(is_leap_year(2000));
    assert(is_leap_year(2400));
}
• prefer initialization to assignment

```c
int count;       // ambiguous
count = 0;       // ambiguous

int count = 0;  // clearer
```

• don't explicitly compare against true/false

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

```c
if (has_prefix("is")) ...
```
• avoid redundant use of true/false

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

this version is very "solution focused"

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

```cpp
bool is_even(int value) {
    return value % 2 == 0;
}
```
• make inter-statement dependencies explicit

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

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

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

bool some_func(int value) {
    if (value % 2 == 0)
        return alpha();
    else
        return beta();
}
• In general, beware of boolean literals

Again, wordy, verbose

```
if (oldest)
    if (last_access < cut_off)
        return true;
    else
        return false;
else
    return false;
```

Refactor

```
return oldest && last_access < cut_off;
```

Much simpler, reads well
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

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

we didn't find it.......... or .................. we found it
(short-circuiting)
• the negation of the termination condition is the iteration's continuation condition
  • !(at == end || values[at] == find)
  • at != end && values[at] != find

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

at++;  

short-circuiting protects values[at]

• then simply fill in the loop body 😊
• 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

```c
typedef struct date * date;

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

```c
/* date.h */

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

what is `const`?

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

is it the date object pointed to?

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

or is it the pointer?

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

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

date.h

struct date;

bool date_equal(const struct date * lhs,  
                const struct date * rhs);
a typedef does **not** create a new type

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

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

consider using a wrapper type instead...

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

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!

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

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

void weak(void)
{
    suit trumps = winter;
}
```
```c
typedef struct { int value; } suit;
extern const suit clubs, diamonds, hearts, spades;

typedef struct { int value; } season;
extern const season spring, summer, autumn, winter;

void strong(void)
{
    suit trumps = winter;
}

const suit clubs = { 0 },
diamonds = { 1 },
hearts = { 2 },
spades = { 3 };
```
• 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
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

```c
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.
in this statement...

- where are the sequence points?
- where are the side-effects?
- is it undefined?

```c
volatile int m;

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

```c
#include "grommit.h"
#include "flange.h"
...
typedef struct {
    grommit w;
    flange f;
    ...
} wibble_t;
```

```c
#include "sink.h"
#include "washer.h"
...
typedef struct {
    sink dest;
    washer w;
    ...
} grommit;
```
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

```
#include <wibble.h>

typedef struct wibble_tag wibble;

wibble * wopen(const char * filename);
int wclose(wibble * stream);
```

- minimal #includes
- not defined
- *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

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

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

```c
int main(int argc, char * argv[])
{
    wibble * w = wibbles.open(argv[1]);
    ...
    wibbles.close(w);
}
```
#ifndef WIBBLE_INCLUDED
#define WIBBLE_INCLUDED
...
...
typedef struct wibble_tag wibble;

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

```c
#include "wibble.h"

void client(void)
{
    wibble * pointer;
    ...
    wibble value;
    ...
    ptr = malloc(sizeof(*ptr));
}
```
shadow data type

- an ADT can declare its size!
- clients can now allocate the memory
- true representation remains abstract

```c
typedef struct wibble
{
  unsigned char size[16];
} wibble;

bool wopen(wibble *, const char *);
void wclose(wibble *);

#include "wibble.h"

void client(const char * name)
{
  wibble w;
  if (wopen(&w, name))
    ...}
  wclose(&w);
}
```
• implementation needs to...
  • define the true representation type

The analogy is that the true type casts a shadow which reveals only its size.

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

typedef struct
{
    grommit g;
    flange f;
    ...
} wibble_rep;

typedef struct
{
    unsigned char size[16];
} wibble;
```

Still some problems though…
• 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

```c
typedef struct wibble {
    ...
} wibble;

typedef struct {
    ...
} wibble_rep;

assert(sizeof(wibble) >= sizeof(wibble_rep))
```
• use a compile time assertion
  • you cannot declare an array with negative size

```c
#include "compile_time_assert.h"

COMPILE_TIME_ASSERT(ok, 1 == 1);

COMPILE_TIME_ASSERT(
  your_message,
  1 == 0);
```

```
gcc
  error: size of array 'your_message' is negative
```
#include "wibble.h"
#include "flange.h"
#include "grommet.h"
#include "compile_time_assert.h"

typedef struct{
    grommet g;
    flange f;
} wibble_rep;

COMPILE_TIME_ASSERT(
    sizeof_wibble_not_less_than_sizeof_wibble_rep,
    sizeof(wibble) >= sizeof(wibble_rep));

...
• the two types must be alignment compatible
  • this means the first member of both types must be alignment compatible

wibble.h

typedef struct wibble
{
    unsigned char size[16];
} wibble;

wibble.c

typedef struct
{
    grommit g;
    flange f;
    ...
} wibble_rep;
use a union to force alignment

alignment.h

```c
typedef union
{
    char c,*cp; 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,...);
} alignment;
```

wibble.h

```c
#include "alignment.h"
...
typedef union wibble
{
    alignment universal;
    unsigned char size[16];
} wibble;
...
```

List all the primitive types in here. One must have the strictest alignment.
• unions are much rarer than structs
  • design is as much about use as implementation

6.7.2.3 Tags
Paragraph 1, sentence 2
Where two declarators that use the same tag declare the same type, they shall both use the same choice of struct, union, or enum.

client.h

typedef struct wibble wibble;
• put a union object inside a struct!
  • drop the union tag name

```c
... typedef struct wibble
{
  union
  {
    alignment universal;
    unsigned char size[16];
  } shadow;
} wibble;
...```

6.7.2.3 Tags
Paragraph 4, sentence 2
Each declaration of a structure, union, or enumerated type which does not include a tag declares a distinct type
- conversion can be via memcpy
- helper function allows assignment

wibble.c

```c
static inline wibble shadow(wibble_rep * src) {
    wibble dst;
    memcpy(&dst, src, sizeof(*src));
    return dst;
}

bool wopen(wibble * w, const char * name) {
    wibble_rep rep = {
        .g = ...,  
        .f = ...,
    };
    *w = shadow(&rep);
    ...
}  
```
• conversion can be via pointer casts
  • helper function allows → operator

wibble.c

```c
static inline wibble_rep * rep(wibble * w) {
    return (wibble_rep *)w;
}

void wclose(wibble * w) {
    rep(w)->g = ...;
    rep(w)->f = ...;
    ...
}
```
Contact

• This course was written by

Jon Jagger
jon@jaggersoft.com
www.jaggersoft.com

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

{ JSL }