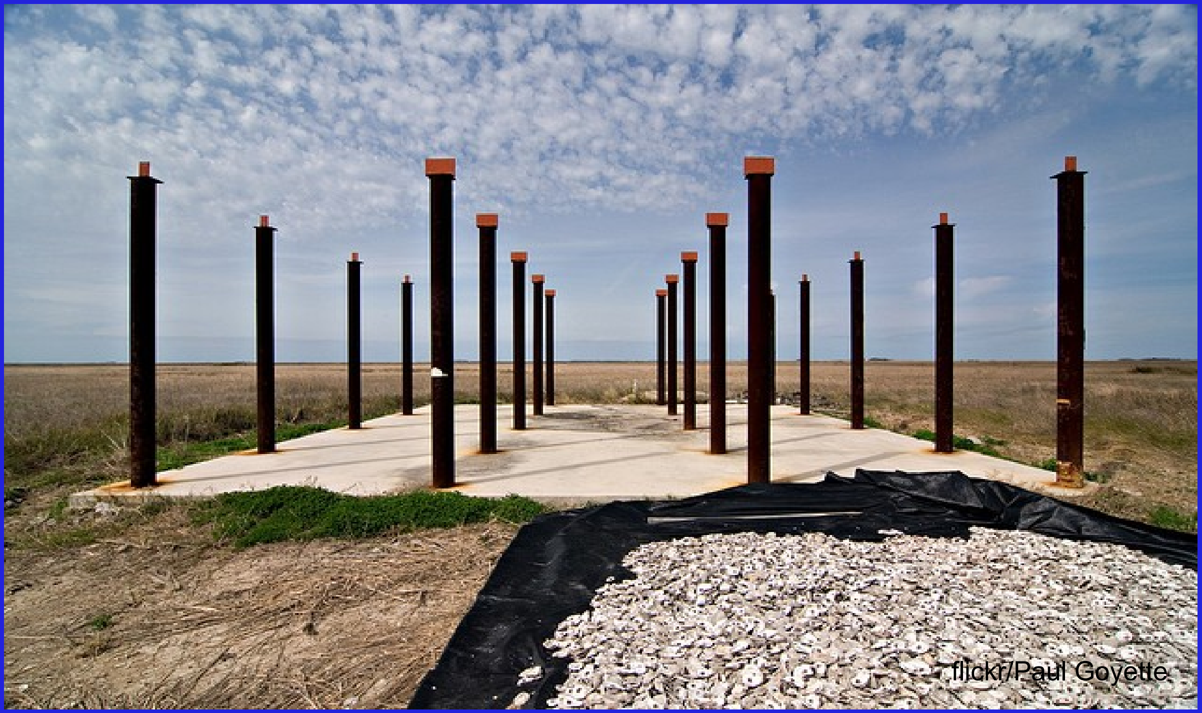
C++ Foundation



Introduction to C++

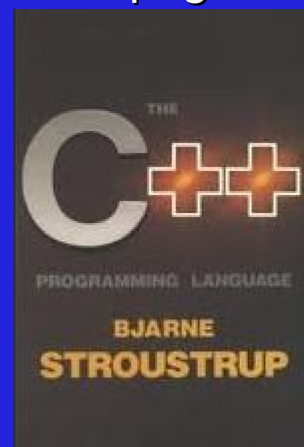
Introduction to C++

- history and use
- sequence points
- evaluation order
- undefined behaviour
- where and why to use C++
- spirit of C++
- hello C++

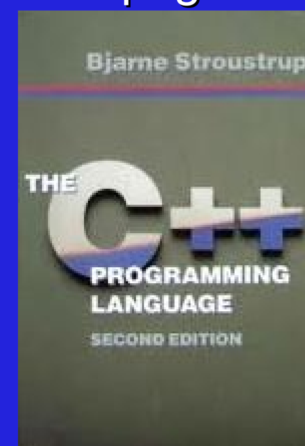
History

- The 3 ages of C++ correspond to editions of The C++ Programming Language book
- 1st age, 1985
- 2nd age, 1991
- 3rd age, 1997
- 4th age, ????
  - C++0x (sic)
  - promises many new language features and libraries

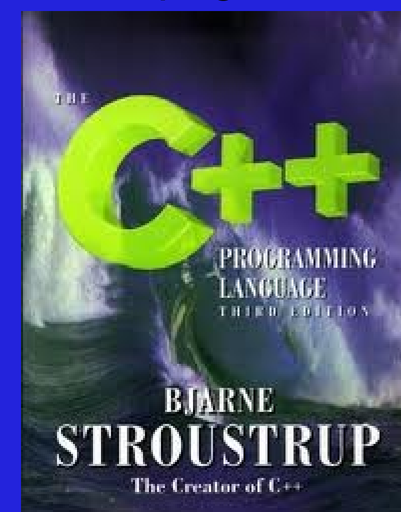
331 pages



720 pages



1040 pages



C/C++ Use

- C and C++ continue to be popular languages
- Tiobe index, language ranking
  - http://www.tiobe.com

	1986	1996	2006	2011
C++	8	3	3	3
C	1	1	2	2

Undefined Behaviour

- Conformance
  - If a “shall” or “shall not” requirement... is violated the behavior is undefined. (4)
- Undefined behavior
  - Behavior, upon use of a nonportable or erroneous program construct or of erroneous data, for which this International Standard imposes no requirement. (3.4.3)

Evaluation Order

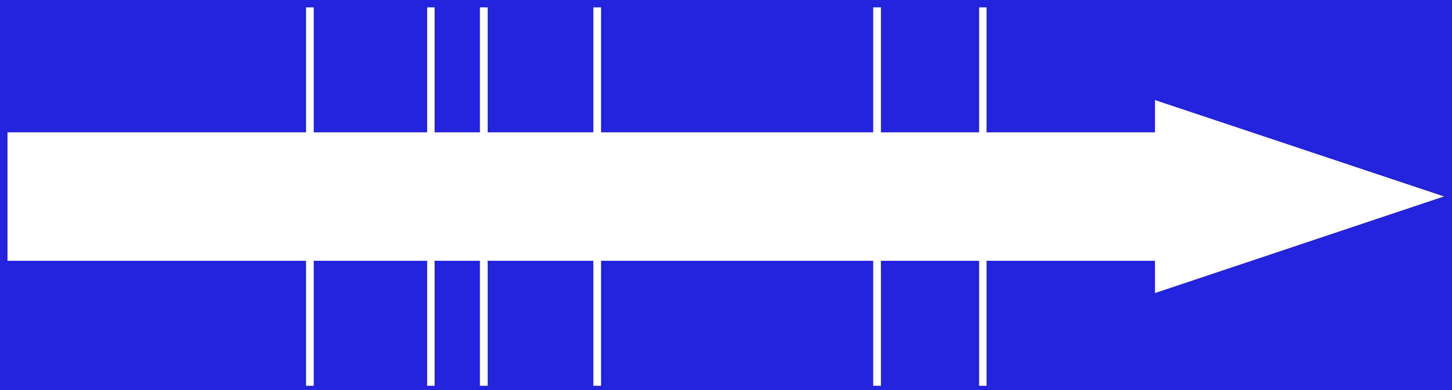
- Evaluation order is not always determined by the layout order of the code
  - statements, initialisers, operands of the short-circuiting and comma operators are evaluated in strict sequence... and that's pretty much it
  - can affect expression certainty and correctness

```
a = f() *  g() + h();
a = f() * (g() + h());
```

there is no required difference in evaluation order between these two assignments

Sequence Points

- 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 (5.1.2.3)



Sequence Points

- sequence points occur...
  - at the end of a full expression
  - after the first operand of these operators
    - && logical and
    - || logical or
    - ?: ternary
    - , comma
  - after evaluation of all arguments and function expressions in a function call
  - at the end of a full declarator
  - that's it!

Sequence Point Rules

- Rule 1
  - Between the previous and next sequence point an object *shall* have its stored value modified at most once by the evaluation of an expression (6.5)
- Rule 2
  - Between the previous and next sequence point... the prior value *shall* be read only to determine the value to be stored (6.5)

```
n = n++
```

Violates rule 1 - undefined

```
n + n++
```

Violates rule 2 - undefined

Multi Paradigm?

- C++ supports many many different programming styles
  - traditional C style
  - data abstraction
  - operator overloading
  - object oriented
  - generic programming (templates)
  - multiple inheritance
  - exceptional control flow
- Very very easy to make a big bad mess
  - keep it simple, focus on a few styles only

Comparing Languages

- Where to use C++?
  - only C and C++ compiler is available
- Why use C++?
  - you need a higher level of abstraction than C
- What to compare with?
  - Compare with Java, C#, etc
  - Don't compare C++ with C

Spirit of C++

- trust the programmer
- speed trumps portability
- stay close to the hardware
- be as close as possible to C, but no closer
- don't pay for what you don't use
- catch errors at compile time
- avoid the preprocessor

Hello C++

- traditional first program

preprocessor #include
(note no .h)

<< streaming operator

“string literal”

```
#include <iostream>

int main()
{
   std::cout << “Hello, world\n”;
}
```

std namespace

:: scope resolution operator

Key

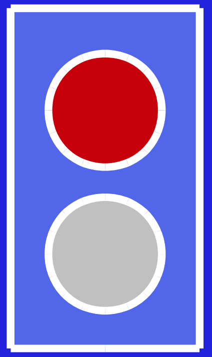
✓ compiles

? compiles but questionable

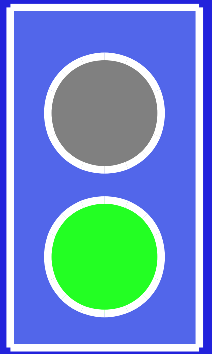
✗ doesn't compile

a bug

a note

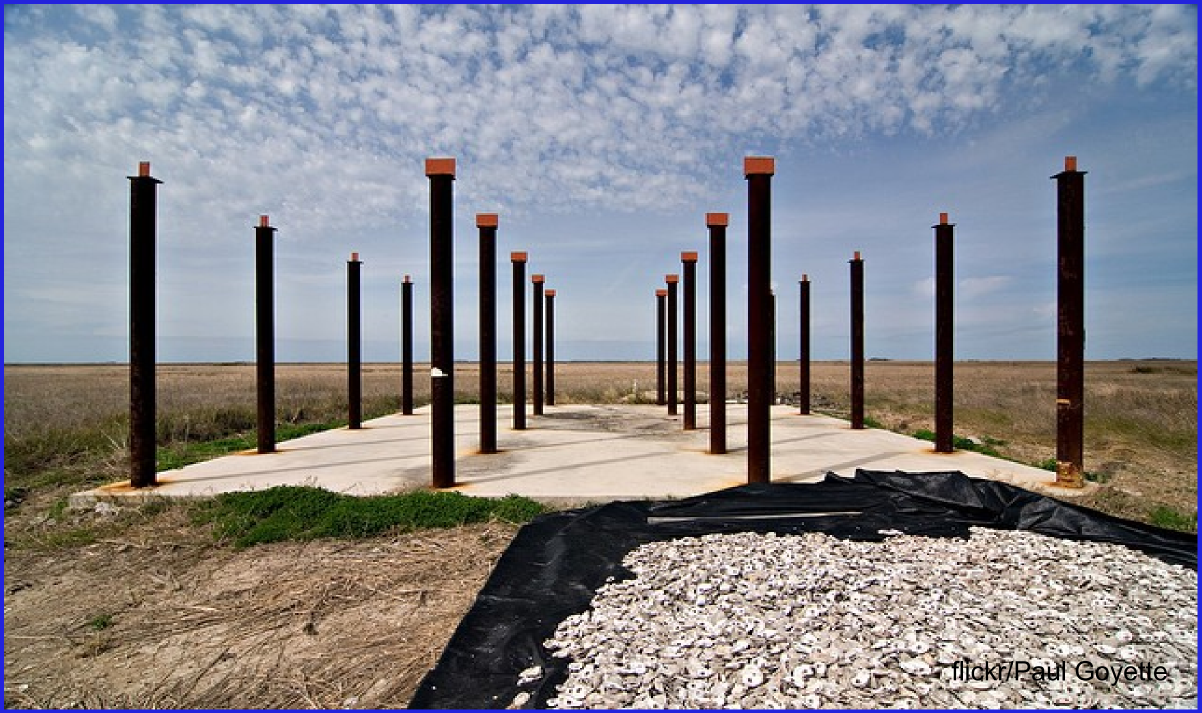


tests fail



tests pass

C++ Foundation



A Brief Tour of C++

A Brief Tour of C++

- By example...

Your task is to create an LCD string representation
of an integer value using a 3x3 grid of space,
underscore, and pipe characters for each digit.
Each digit is shown below (using a dot instead of
a space)

```
._.  ...  ._.  ._.  ...  ._.  ._.  ._.  ._.  ._.
|.|  ..|  ._|  ._|  |_|  |_.  |_.  ..|  |_|  |_|
|_|  ..|  |_.  ._|  ..|  ._|  |_|  ..|  |_|  ..|
```

Example: `910`

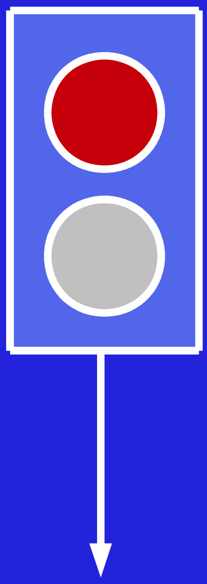
```
._. ... ._.
|_| ..| |.|
..| ..| |_|
```

Test First Development

- Think of a test as an executable specification

```
...
int main()
{
    lcd_spec(0, lcd(
        “ _ “,
        “| |”,
        “|_|”
    ));

    std::cout << “All passed”
              << std::endl;
}
```

lcd_tests.cpp

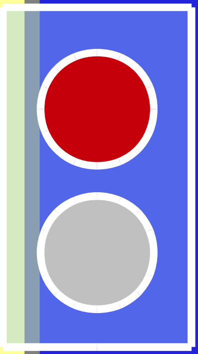


A test that doesn't compile yet certainly counts as a failing test

Test First Development

- Our tests use this helper function

```
#include <string>
#include <vector>

typedef std::vector<std::string> lcd_grid;

lcd_grid lcd(string s1, string s2, string s3)
{
    lcd_grid result;
    result.push_back(s1);
    result.push_back(s2);
    result.push_back(s3);
    return result;
}
```

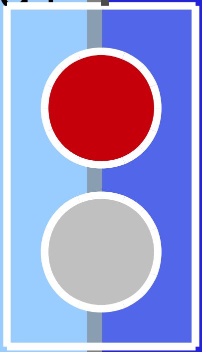


lcd.cpp

Test First Development

lcd_tests.cpp

```
#include “lcd.hpp”
#include <iostream>
...
void lcd_spec(int value, lcd_grid grid)
{
    std::string expected = to_string(grid),
                actual = to_string(lcd(value));
    if (expected != actual)
    {
        std::cerr
            << “lcd(“ value << “)” << std::endl
            << “expected== << std::endl
            << expected << std::endl
            << “actual==” << std::endl
            << actual << std::endl;
        std::exit(EXIT_FAILURE);
    }
}
```



Test First Development

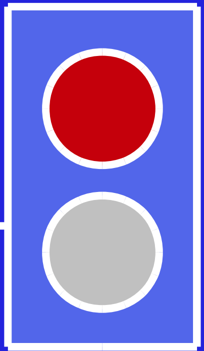
- Get the tests to compile and link

lcd.hpp

```
#ifndef LCD_INCLUDED
#define LCD_INCLUDED

#include <string>
#include <vector>

typedef std::vector<std::string> lcd_grid;

lcd_grid lcd(int value);

#endif
```

lcd.cpp

```
#include “lcd.hpp”

lcd_grid lcd(int value)
{
    throw “to do”;
}
```

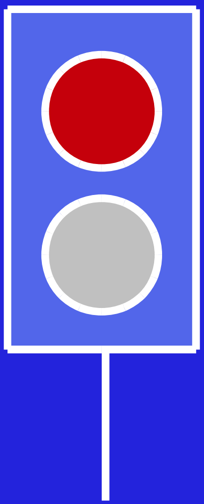


Test First Development

- The tests now run, but fail, the perfect start :-)

```
...
int main()
{
    lcd_spec(0, lcd(
        “ _ “,
        “| |”,
        “|_|”
    ));
}
```

lcd_tests.cpp



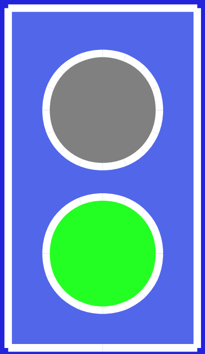
```
$g++ -Wall -Wextra lcd*.cpp && ./a.out
terminate called after thowing an instance
of char const *
```

Test First Development

- Make the tests pass

```
const lcd_grid digits[] =
{
    lcd(“ _ “,
        “| |”,
        “|_|”
    ),
};

lcd_grid lcd(int value)
{
    if (value == 0)
        return digits[0];
    else
        throw “to do”;
}
```

lcd.cpp

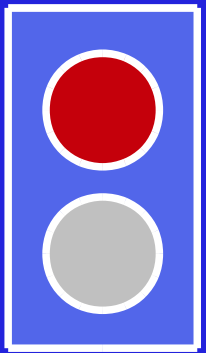


Test First Development

- Write another failing test

```
...
int main()
{
    ...
    #define WS “ “

    lcd_spec(12, lcd(
        “   “ WS “ _ “,
        “  |” WS “ _|”,
        “  |” WS “|_ ”
    ));

    #undef WS
}
```

lcd_tests.cpp

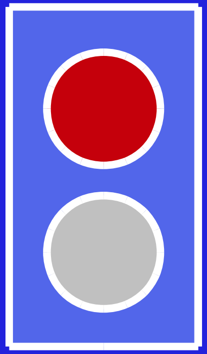


Test First Development

- Make the tests pass

```
const lcd_grid digits[] =
{
    lcd(“ _ “,
        “| |”,
        “|_|”
    ),
    lcd(“   “,
        “  |”,
        “  |”
    ),
    lcd(“ _ “,
        “ _|”,
        “|_ “
    ),
};
```

lcd.cpp

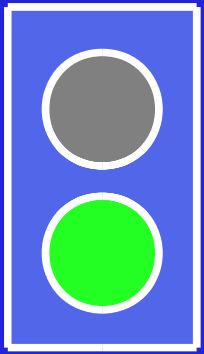


Test First Development

- Get the test to pass

```
lcd_grid lcd::grid(int value)
{
    if (value < 10)
        return digits[value];
    else
    {
        lcd_grid lhs = lcd(value / 10);
        lcd_grid rhs = digits[value % 10];
        return lcd(
            lhs[0] + “ “ + rhs[0],
            lhs[1] + “ “ + rhs[1],
            lhs[2] + “ “ + rhs[2]);
    }
}
```

lcd.cpp

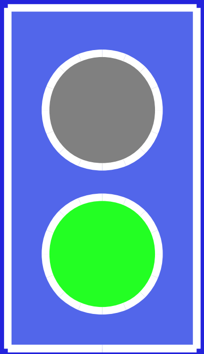


Test First Development

- Refactor when at green

```
lcd_grid lcd::grid(int value)
{
    if (value < 10)
        return digits[value];
    else
    {
        lcd_grid lhs = lcd(value / 10),
                 rhs = digits[value % 10];
        const std::string ws = “ “;
        return lcd(
            lhs[0] + ws + rhs[0],
            lhs[1] + ws + rhs[1],
            lhs[2] + ws + rhs[2]);
    }
}
```

lcd.cpp



What Did We Use?

- std::string - to abstract away char* horribleness

```
class string
{
public:
    string();
    string(const char *);
    string(const string &);
    ~string();
    ...
};
```

construct an empty string

construct a string from a '\0' terminated array of chars

construct a string from another string

destruct a string

simplified (string is actually a typedef)

What Did We Use?

- std::string - to abstract away char* horribleness

```
bool operator==(string lhs, string rhs);
bool operator!=(string lhs, string rhs);

string operator+(string lhs, string rhs);
```

```
string expected = ...,
       actual = ...;

if (expected == actual)...
```

```
return lcd(lhs[0] + ws + rhs[0],
           lhs[1] + ws + rhs[1],
           lhs[2] + ws + rhs[2]);
```

What Did We Use?

- std::vector<> - a resizeable array

```
template<typename Type>
class vector
{
public:
    vector();
    vector(const vector &);
    ~vector();
    ...
};
```

construct an empty vector

construct an a vector as a copy of another vector

destruct a vector

simplified

What Did We Use?

- std::vector<> - a resizeable array

```
template<typename Type>
class vector
{
public:
    void push_back(Type pushed);
    ...
    Type & operator[](size_t at);
    ...
};
```

```
std::vector<std::string> result;
result.push_back(s1);
```

```
std::vector<std::string> lhs = ...;
std::vector<std::string> rhs = ...;
    lhs[0] ...  rhs[0]
    lhs[1] ...  rhs[1]
```

What Did We Use?

- std::ostream - an output stream

```
class ostream
{
    ...
};

extern ostream cerr;
extern ostream cout;

ostream & endl(ostream &);
```

tied to stderr from C

tied to stdout from C

'\n' and flush

simplified

What Did We Use?

- std::ostream - an output stream

```
class ostream
{
public:
    ostream & operator<<(string);
    ostream & operator<<(int);
    ostream & operator<<(const char *);
    ...
};
```

```
std::string expected = ...;
std::cerr << "lcd("
          << value
          ...
          << expected
          ...
```

C++ Foundation

Functions, Operators, and Data Structures

Functions, Operators, and Data Structures

- member functions
- private access
- overloading
- default arguments
- function templates
- references
- passing arguments
- operators

Member Data and Member Functions

- C++ structs and classes can contain functions

C

```
struct date
{
      int year;
      int month;
      int day;
};
int day_number(const date *);
```

```
void eg(struct date when)
{
    when.day = 30;
    day_number(&when);
}
```

C++

```
struct date
{
      int year;
      int month;
      int day;

      int day_number() const;
};
```

```
void eg(date when)
{
    when.day = 30;
    when.day_number();
}
```

Access Control

- C++ structs and classes can control access using the public and private keywords

```
struct date
{
public:
      ...
      int year;

private:
      ...
      int month;
};
```

```
date when;

when.year;

when.month;
```

No Abstraction

- C has limited scope for abstraction based on use
  - C structs typically expose their representation
  - An open invitation to access their "private parts"! not safe! not polite!

```
struct date
{
      int year, month, day;
};
```

```
void oops(struct date * when)
{
    when->day = 42;
}
```

A Model of Politeness

- C++ structs and classes offer a model *not* based on representation
  - Lights, camera, abstraction!

```
struct date
{
        ...
        int year() const;
        int month() const;
        int day() const;
        ...
};
```

```
void eg(date when)
{
      today.day()
}
```

don't forget the parentheses!

What the heck does that const mean?

```
struct date
{
      ...
      int day() const;
      ...
};
```

a const at the end of a member function declaration promises...

```
today.day()
```

...that if you call that member function on an object...

```
date today;
```

...the member function's definition won't change the visible state of that object

How to Define Member Functions?

fubar.h

```
int f();
```

fubar.c

```
#include "fubar.h"

int f()
{
      ...
}
```

C

fubar.hpp

```
struct fubar
{
   int f();
   ...
};
```

fubar.cpp

```
#include "fubar.hpp"

int fubar::f()
{
      ...
}
```

C++

The Scope Resolution Operator

- Member function declarations require a matching member function definition

```
#include “date.hpp”

int date::year() const
{
     ...
}
```

don't forget the #include

:: is called the scope resolution operator

```
struct date
{
      ...
      int year() const;
      ...
};
```

the const must be repeated on the member function definition

Private Access

- Only member functions have access to private members
- All members are accessible to all other members through the implicit this pointer

```
struct date
{
     ...
     int year() const;
     ...
private:
     long time_stamp;
};
```

```
#include "date.hpp"

int date::year() const
{
     ... time_stamp ...
}
```

Overloading

- Functions with the same name and in the same scope are said to *overload* each other
- Function parameters must differ somehow
- A difference in return type alone is not sufficient

```
int    min(int    lhs, int    rhs);
long   min(long   lhs, long   rhs);
double min(double lhs, double rhs);
```

✔

?

```
int    random();
long   random();
double random();
```

✘

```
random();
```

♪ overloading != overriding

Overloading

- The compiler resolves a call to the overload with the best argument-parameter match
- Resolution must be unambiguous

```
int    min(int    lhs, int    rhs);
long   min(long   lhs, long   rhs);
double min(double lhs, double rhs);
```

```
min(2.5, 4.5)
min(2, 4)
```

✓

```
min(2, 4.5)
min(2.5, 4)
```

✗

Internal Default Arguments

- Overloading and forwarding can provide “internal” defaults

```
void write(int number, std::ostream & to);
void write(int number);
```

```
void write(int number)
{
    write(number, std::cout);
}
```

```
write(42);
write(42, std::cerr);
```

External Default Arguments

- An explicit “external” =default syntax also exists

```
void write(int number,
           std::ostream & to = std::cout);
```

```
write(42);
```

compiler rewrites to...

```
write(42, std::cout);
```

```
write(42, std::cerr);
```

the default is written on the declaration, not on the definition

defaults are written on the rightmost parameters

External Defaults...?

- Can increase header dependencies
- Can be surprising - best avoided

```
coord make_coord(int x = 0, int y = 0);
```

```
make_coord();          make_coord(0,0);
```

```
make_coord(2,5);       make_coord(2,5);
```

```
make_coord(42);        make_coord(42,0);
```

?!

Function Templates

- What about when functions differ in their *types* but not their definition?

```
int min(int lhs, int rhs)
{
  return lhs < rhs ? lhs : rhs;
}
```

```
double min(double lhs, double rhs)
{
  return lhs < rhs ? lhs : rhs;
}
```

```
?? min(?? lhs, ?? rhs)
{
  return lhs < rhs ? lhs : rhs;
}
```

Function Templates

- The compiler can use a function template to generate functions to match function calls!

```
template<typename T>
T min(T lhs, T rhs)
{
  return lhs < rhs ? lhs : rhs;
}
```

```
min(4,2)
```

typeof(4) == typeof(2) == int

T == int

```
int min(int lhs, int rhs)
{
  return lhs < rhs ? lhs : rhs;
}
```

Template Type Names

- It is common to express the template type's requirements in its name

```
template<typename T>
T min(T lhs, T rhs)
{
  return lhs < rhs ? lhs : rhs;
}
```

s/T/Comparable/

```
template<typename Comparable>
Comparable min(Comparable lhs, Comparable rhs)
{
  return lhs < rhs ? lhs : rhs;
}
```

Function Templates

- Can have more than one templated type

```
template<typename Iterator,
         typename Function>
Function for_each(Iterator at,
                  Iterator end,
                  Function f)
{
    for (; at != end; ++at)
        f(*at);
    return f;
}
```

```
const char * str = “Hello”;
for_each(str, str + 5, std::putchar);
```

Function Template Diagnostics

- Can be *hard* to understand!

```
void wtf()
{   ...
    std::for_each(begin, end, s());
}
```

```
In file included from /usr/include/c++/4.4/algorithm:62
...
/usr/include/c++/4.4/bits/stl_algo.h: In function
  '_Funct std::for_each(_IIter, IIter, _Funct)
  [with __Iter = int*, _Funct = s]'
...perhaps 450 lines of babble...
instantiated from here
/usr/include/c++/4.4/bits/stl_algo.h:4200:
  error: no match for call to '(s) (int&)'
```

References

- Look like pass by copy, but isn't
- An alias to an object

C

```
void f(int value)
{
    value++;
}
void eg()
{
    int x = 42;
    f(x);
    assert(x == 42);
}
```

C

```
void f(int * value)
{
    (*value)++;
}
void eg()
{
    int x = 42;
    f(&x);
    assert(x == 43);
}
```

C++

```
void f(int & value)
{
    value++;
}
void eg()
{
    int x = 42;
    f(x);
    assert(x == 43);
}
```

This is not
f(&x);

const references

- Look like pass by copy, but isn't
- An alias to a readonly object

C

```
void f(int value)
{
    value++;
}
void eg()
{
    int x = 42;
    f(x);
    assert(x == 42);
}
```

C++

```
void f(int & value)
{
    value++;
}
void eg()
{
    int x = 42;
    f(x);
    assert(x == 43);
}
```

C++

```
void f(const int & value)
{
    //value++;
}
void eg()
{
    int x = 42;
    f(x);
    assert(x == 42);
}
```

Operators

- Highly stylized functions
- You can overload most operators

```
int deadline, today;
...
if (deadline == today)...
```

```
struct date { ... };
bool operator==(const date &, const date &);
```

```
date deadline, today;
if (deadline == today)...
```

infix notation

Template Type Requirements

- Template types require a *uniform* syntax

```
template<typename T >
T min(T lhs, T rhs)
{
  return lhs < rhs ? lhs : rhs;
}
```

this definition requires T supports the < operator

```
struct date { ... };
bool operator<(const date &, const date &);
```

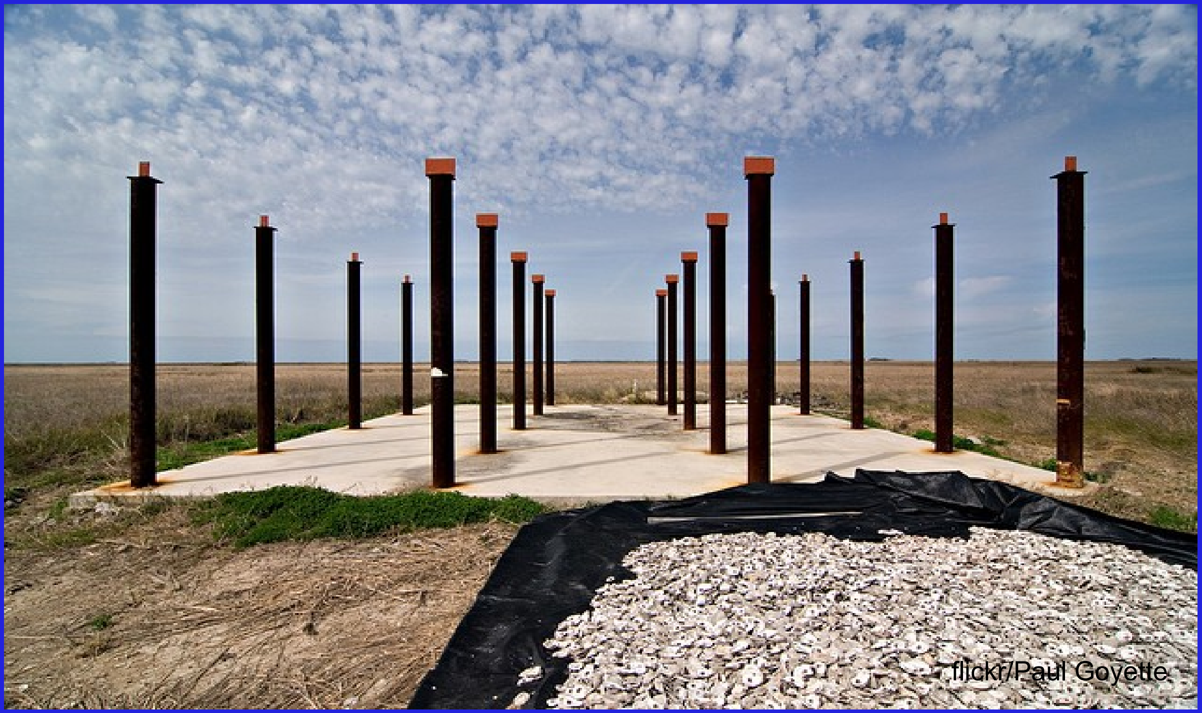
```
date earliest,estimate;
...
earliest = min(earliest, estimate);
```

✓

Parameter Passing

- By const & to mimic pass by copy for non-primitive types is a common idiom
- By copy when the argument is an <algorithm> parameter is also common
- By non-const & when the function modifies the argument
- By copy when the argument is a primitive type (eg int, bool, enum)
- By plain pointer to indicate object lifetime considerations

C++ Foundation



Objects, Classes, and Lifetimes

Objects, Classes, and Lifetimes

- constructors
- destructors
- the three storage classes
- new expressions
- delete expressions
- copy construction
- copy assignment
- smart pointers
- copying rule of three

Construction

- A constructor has the same name as the class

```
#include "date.hpp"

void example()
{
    date xmas(2011, 12, 25);
    assert(xmas.day() == 25);
}
```

date.hpp

```
class date
{
public:
    date(int year, int month, int day);
    ...
};
```

Construction

- Define constructors using the
  : member(*initialization*), syntax

date.hpp

```
class date
{
public:
    date(int year, int month, int day);
    ...
private:
    long time_stamp;
};
```

date.cpp

```
#include "date.hpp"

date::date(int year, int month, int day)
  : time_stamp(calc(year,month,day))
{

}
```

you *cannot* do initialization here

No Defaults

- Primitive data members *don't* acquire a default value - just like they don't in C

```
date::date(int year, int month, int day)
  // : time_stamp(...)
{
}
```

initialization commented out

```
void no_defaults()
{
    date xmas(2011, 12, 25);
    std::cout << xmas.time_stamp;
}
```

-740281080

assumes time_stamp is publicly accessible

Default Construction

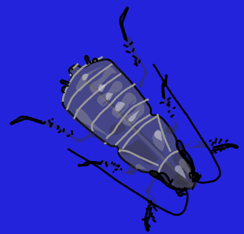
- Value type objects can often be created without any arguments
  - If all arguments have defaults the effect is the same

```
class rope
{
public:
    rope();
    ...
};
```

```
#include “rope.hpp”

void example()
{
  rope old;
  ...
}
```

```
class rope
{
public:
    rope(const char * = “”);
    ...
};
```

Very Common Mistake



```
void correct()
{
  rope old;
  ...
}
```

this defines an object called old, of type rope

```
void incorrect()
{
  rope old();
  ...
  old.member
}
```

this declares a *function* called old, that accepts no arguments and returns a rope object

Destruction

- When an object's life ends it's destructor is *automagically* called
- The ~ “complement” of a constructor

```
#include “date.hpp”

void example()
{
    date xmas(2011, 12, 25);
    ...
} // xmas.~date()
```

```
class date
{
public:
    ~date();
    ...
};
```

the destructor never has any parameters - *you* “never” call it...

3 Types of Storage

- static - eg global variables
- automatic - eg local variables
- dynamic - eg new/delete

```
date a(2011,3,7);
static date b(2011,3,29);

date example(date c)
{
    static date d(2011,3,7);
    date e(2011,3,7);
    date * f = new date(2011,3,7);
    ...
    delete f;
    return date(2011,12,25);
}
```

a	
b	
c	
d	
e	
f	
*f	
return	

fill in the 8 storage classes

3 Types of Storage

- static - eg global variables
- automatic - eg local variables
- dynamic - eg new/delete

```
date a(2011,3,7);
static date b(2011,3,29);

date example(date c)
{
    static date d(2011,3,7);
    date e(2011,3,7);
    date * f = new date(2011,3,7);
    ...
    delete f;
    return date(2011,12,25);
}
```

a	static
b	static
c	automatic
d	static
e	automatic
f	automatic
*f	dynamic
return	automatic

fill in the 8 storage classes

Static Storage

- The order of initialization within a translation unit is defined: top to bottom, left to right
- The order of initialization across translation units is *undefined*

```
date global(2011,3,7);

void function()
{
   ...
}

// global.~date()
```

global's constructor is usually called *before* function invocation

global's destructor may be called at program termination

Automatic Storage

- A parameter object or local object is automatically destructed when it goes out of scope

```
void eg(date param)
{
    for (...)
    {
        date local;
        ...
        ...
        // local.~date()
    }
    // param.~date()
}
```

Dynamic Storage

- Create a dynamic object with a *new* expression
  - Allocates memory dynamically, like malloc()
  - Constructs an object in that memory
  - Returns a strongly typed pointer to the object

new keyword

```
void example()
{
    date * xmas = new date(2011, 12, 25);
}
```

object construction

Dynamic Storage

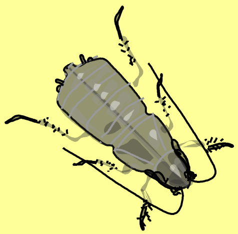
- When a pointer goes out of scope *nothing* happens - the object pointed to does *<u>not</u>* have its destructor automatically called
- Destroy a new'd object with a delete expression
  - Calls the destructor on the pointed to object
  - Releases the dynamic memory - like free()

```
void example()
{
    date * xmas = new date(2011, 12, 25);
    ...
    delete xmas;                // xmas->~date();
                                // free(xmas);
}
```

Puzzle

- Spot the bug
- What happens in this case?

```
#include “B.hpp”

class fubar
{
public:
    fubar(int size) : elems(new B[size]) {}
    ~fubar() { delete elems; }
    ...
private:
    B * elems;
};
```



Solution

- delete[ ] calls all the destructors (in reverse) and then releases the memory

```
#include “B.hpp”

class fubar
{
public:
    fubar(int size) : elems(new B[size]) {}
    ~fubar() { delete[] elems; }
    ...
private:
    B * elems;
};
```

Consider using <vector> instead

Copying

- Copying an object...
  - As it is created is called *copy construction*
  - After it is created is called *copy assignment*

```
void example()
{
    date xmas(2011,12,25);

    date copy(xmas);

    copy = xmas;

    date another = xmas;
}
```

copy construction

copy assignment

copy construction

Copying

```
class date
{
public:
    ...
    date(const date & other);
    date & operator=(const date & rhs);
    ...
};
```

```
date::date(const date & other)
    ...
{
}

date & date::operator=(const date & rhs)
{   ...
}
```

Copy Construction

- The order of initialization is defined by the order of declaration

```
date::date(const date & other)
     : year(other.year)
     , month(other.month)
     , day(other.day)
{
}
```

```
class date
{
     ...
private:
     int year;
     int month;
     int day;
};
```

Why?

Copy Construction

- If you don't write a copy constructor the compiler will try and write one for you
  - It will do a member-by-member copy construction

```
/*
date::date(const date & other)
    : year(other.year)
    , month(other.month)
    , day(other.day)
{
}
*/
```

Copy Assignment

- If you don't write a copy assignment operator the compiler will attempt to write one for you
  - It will do a member-by-member copy assignment

```
/*
date & date::date(const date & rhs)
{
    year = rhs.year;
    month = rhs.month;
    day = rhs.day;
    return *this;
}
*/
```

Smart Pointers

- The dereference and arrow operators can be overloaded; an object can be designed to look like, feel like and smell like a plain pointer - but act smarter

```
class wibble_ptr
{
public:
    ...
    wibble & operator*() const;
    wibble * operator->() const;
    ...
private:
   wibble * raw;
};
```

Smart Pointers

- For example, consider a null-dereference checking smart pointer

```
void raw(wibble * ptr)
{
    ptr->use();
}
```

what if ptr is null...

```
void smart(wibble_ptr ptr)
{
    ptr->use();
}
```

compiler rewrites...

```
void smart(wibble_ptr ptr)
{
    ptr.operator->()->use();
}
```

Smart Pointers

```
wibble * wibble_ptr::operator->() const
{
    if (!raw)
        ...

    return raw;
}
```

throw an exception
(covered later)

Pointer Parameters

- Good design is often associated with fewer bald pointers and more smart pointers
- Aim to make ownership and lifetime an explicit part of design
- Assume reference parameter objects do not live beyond the end of the function call - don't store their address

Copying - 3 Options

- 1. Let the compiler write them and add a comment to that effect

```
class date
{
public:
    date(int year, int month, int day);
    // compiler generated copy c'tor ok
    // compiler generated copy assignment ok
    ~date();
    ...
private:
    long yyyymmdd;
};
```

Copying - 3 Options

- 2. Write your own because the compiler generated ones would be wrong

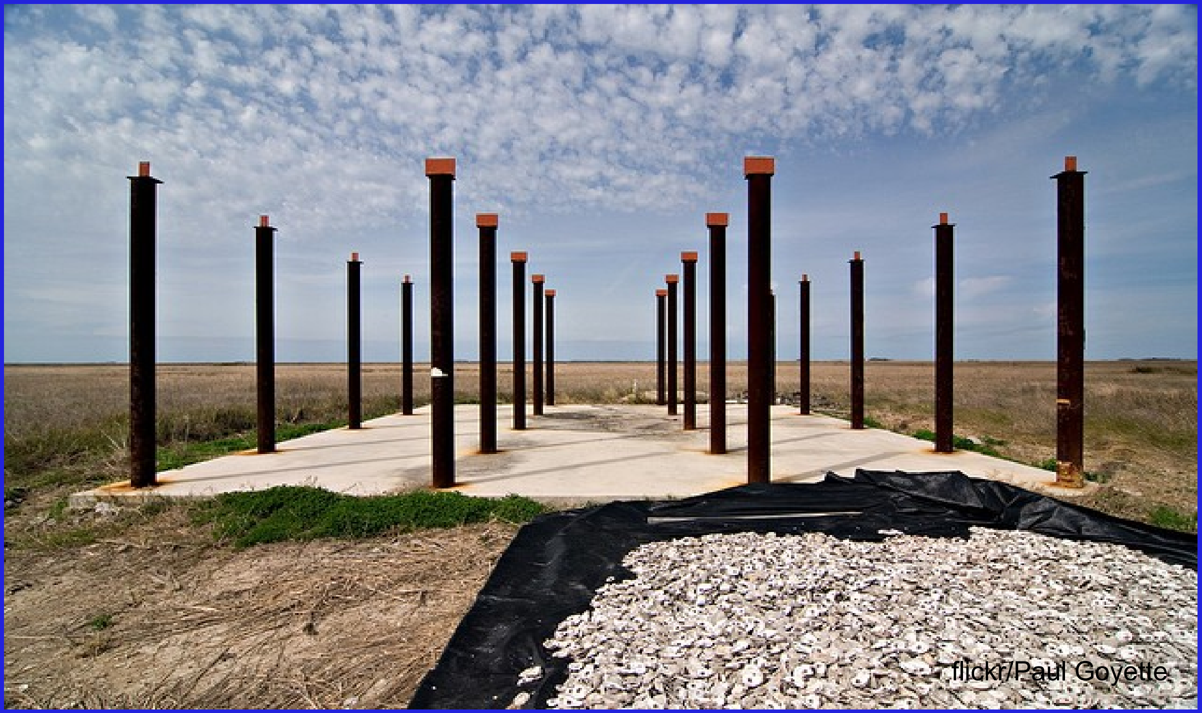
```
template<typename Type>
class shared_ptr
{
public:
    shared_ptr(Type * ptr);
    shared_ptr(const shared_ptr &);
    shared_ptr & operator=(const shared_ptr &);
   ~shared_ptr();
    ...
private:
    Type * raw;
    unsigned int * count;
};
```

Copying - 3 Options

- 3. Turn copying off by declaring them private
  - Since they are never used they don't need to be defined

```
template<typename Type>
class scoped_ptr
{
public:
    scoped_ptr(Type * ptr);
   ~scoped_ptr();
    ...
private: // inappropriate
    scoped_ptr(const scoped_ptr &);
    scoped_ptr & operator=(const scoped_ptr &);
private:
    Type * raw;
};
```

C++ Foundation



Control Flow, Iterators, and Exceptions

Control Flow, Iterators, and Exceptions

- throwing exceptions
- catching exceptions
- exceptions and object lifetime
- the RAII idiom
- predefined exception classes
- iterators
- iterator pairs to express a range
- iterator based algorithm examples

Puzzle

- Consider a function to find the average of a vector of doubles
- What should this function return if the vector is empty?

```
double average(const std::vector<double> & data);
```

```
void puzzle()
{
    std::vector<double> empty;
    assert(average(empty) == ???);
}
```

Introducing *throw*

- Stops normal “forward” execution
- The program starts to unwind backwards!

<u>not</u> a return
not tied to double

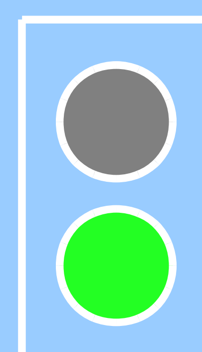
```
double average(const std::vector<double> & data)
{
    if (data.empty())
        throw expression;
    ...
}
```

return double;

Introducing *try* and *catch*

- This test passes if average throws any kind of exception

```
void check_average_of_empty_vector()
{
    std::vector<double> empty;
    bool caught = false;
    try
    {
        average(empty);
    }
    catch (...)
    {
        caught = true;
    }
    assert(caught);
}
```



test fails if no exception is thrown

catch-all
the ... is part of the syntax and not ellision!

Refined test

```
void check_average_of_empty_vector()
{
    std::vector<double> empty;
    bool caught = false;
    try
    {
        average(empty);
    }
    catch (std::invalid_argument &)
    {
        caught = true;
    }
    catch (...)
    {
    }
    assert(caught);
}
```

test fails if no exception is thrown

test passes if invalid_argument is thrown

test fails if different exception is thrown

Standard Exception Classes

- Live in <stdexcept>

```
namespace std
{
    exception;
        bad_cast;
        bad_typeid;
        bad_alloc;
        bad_exception;
        logic_error;
            domain_error;
            invalid_argument;
            length_error;
            out_of_range;
        runtime_error;
            range_error;
            overflow_error;
            underflow_error;
        ...
}
```

thrown by dynamic_cast
thrown by typeid
thrown by new

errors in the internal logical of the program

errors that can only be determined at runtime

The exception Base Class

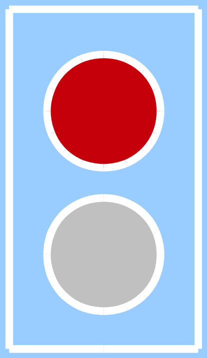
- Lives in <exception>

```
namespace std
{
    class exception
    {
    public:
        exception();
        virtual ~exception();
        exception(const exception &);
        exception & operator=(const exception &);
        virtual const char * what() const;
    private:
        ...
    };
}
```

Refined test

- This test passes only if a *specific* exception is thrown with a *specific* diagnostic string

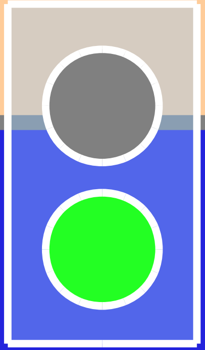
```
void check_average_of_empty_vector()
{
    std::vector<double> empty;
    try
    {
        average(empty);
        assert(false);
    }
    catch (std::invalid_argument & error)
    {
        assert(error.what() == std::string(“empty”));
    }
    catch (...)
    {
        assert(false);
    }
}
```



Average

- Modified to make the test pass

```
#include <stdexcept>

double average(const std::vector<double> & data)
{
    if (data.empty())
        throw std::invalid_argument(“empty”);

    double sum = 0.0;
    for (size_t at = 0; at != data.size(); at++)
        sum += data[at];
    return sum / data.size();
}
```



Object Lifetime

- A fully constructed object *will* have it's destructor called automatically when it goes out of scope - regardless of how it goes out of scope

```
void f()
{
    wibble w;
}
```

```
w.~wibble()
```

```
w.~wibble()
```

```
void f()
{
    wibble w;
    throw ...;
}
```

Resource Acquisition is Initialization

- Acquire a resource in a constructor so you can *automatically* release it in the destructor

```
class auto_file
{
public:
    auto_file(const std::string & name)
        : file(std::fopen(name))
    {
    }
    ~auto_file()
    {
        std::fclose(file);
    }
    ...
private:
    FILE * file;
};
```

```
file.~auto_file()
```

```
void eg(const std::string & name)
{
    auto_file file(name);
    ...exception?...
}
```

```
file.~auto_file()
```

Common Mistakes/Misunderstanding

throwing new'd objects (drop the new)

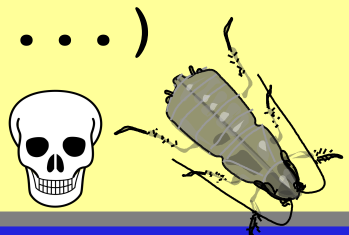
```
throw new std::invalid_argument(“...”);
```

```
catch (std::exception error)
{
    // ...
}
```

catching by copy
(catch by reference)

```
catch (std::exception & error)
{
}
```

```
catch (...)
{
}
```



catching an
exception and
doing nothing?

Iteration

- Two models for iteration...

```
int array[42];
for (int at = 0; at != 42; ++at)
{
      eg(array[at]);
}
```

random access

```
int array[42];
for (int * pos = &array[0];
      pos != &array[42];
      ++pos)
{
      eg(*pos);
}
```

sequential access

Iteration

- C++ iterators follow the sequential model

```
typedef std::list<int> container;

container values;
for (container::iterator pos = values.begin();
     pos != values.end();
     ++pos)
{
     eg(*pos);
     ...
}
```

```
template<typename T>
class list<T>
{    ...
     class iterator
     {    ...
          operator*()
          operator++()
     };
     bool operator==(iterator, iterator);
     bool operator!=(iterator, iterator);
}
```

begin() and end()

- Container classes offer begin() and end() member functions
- end() returns an iterator "*one-beyond-the-end*"

```
template<typename Type>
class list
{   ...
    class iterator { ... };
    class const_iterator { ... };

    iterator begin();
    iterator end();

    const_iterator begin() const;
    const_iterator end() const;
    ...
};
```

begin and end are overloaded on const

Iterator Pair == Range

- Using a pair of iterators to express a range is a dominant C++ idiom
- The standard library offers many algorithms based on iterator pairs

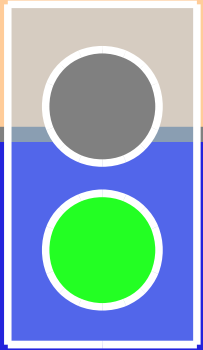
```
template<typename Iterator, typename Value>
Value accumulate(Iterator at, Iterator end,
                 Value sum)
{
    while (at != end)
    {
        sum += *at;
        ++at;
    }
    return sum;
}
```

#include <numeric>

Refactor

- average() implemented using accumulate

```
#include <numeric>

double average(const std::vector<double> & data)
{
    if (data.empty())
        throw std::invalid_argument(“empty”);

    return std::accumulate(
         data.begin(), data.end(), 0.0)
         / data.size();
}
```



std::sort

```
template<typename Iterator>
void sort(Iterator begin, Iterator end);
```

```
#include <algorithm>

void example(std::list<int> & values)
{
    ...
    std::sort(values.begin(), values.end());
    ...
}
```

std::for_each

```
template<typename Iterator, typename Function>
Function for_each(Iterator begin, Iterator end,
                  Function f);
```

```
#include <algorithm>

void print(int value)
{
    std::cout << value << ',';
}

void example(const std::list<int> & values)
{
   std::for_each(values.begin(),
                 values.end(),
                 print);
}
```

C++ Foundation

Program Organization and Dependency Management

Program Organization and Dependency Management

- namespace, “packages”
- using directives and declarations
- explicit qualification
- header files and source files
- unnecessary #includes
- Koenig lookup, argument dependent lookup
- forward declarations
- dependency injection

Namespaces

- A class is not a useful unit of design!
- Collaborating classes are, and can live inside a package, a named scope

grammar_lib/

non_terminal.hpp
production.hpp
production_entry.hpp
terminal.hpp
...

```
namespace grammar_lib
{
    class non_terminal
    {   ...
    };
}
```

```
namespace grammar_lib
{
    class production
    {   ...
    };
}
```

Header Guards

- Always use macro guards to ensure header files are idempotent (beware copy & paste)
- Make header macro guards reflect the folder/namespace name *and* the file/class name

grammar_lib/non_terminal.hpp

```
#ifndef GRAMMAR_LIB_NON_TERMINAL_HPP_INCLUDED
#define GRAMMAR_LIB_NON_TERMINAL_HPP_INCLUDED

...

#endif
```

Using Directives

- Pulls in *all* names into the current scope

```
namespace std
{
    class ostream { ... };
    ostream & endl(ostream &);
    extern ostream cout;
}
```

iostream

```
#include <iostream>

using namespace std;

void eg()
{
   cout << "Hello" << endl;
}
```

using directive

Using Declaration

- Pulls a *specific* name into the current scope

```
namespace std
{
    class ostream { ... };
    ostream & endl(ostream &);
    extern ostream cout;
}
```

iostream

```
#include <iostream>

using std::cout;

void eg()
{
   cout << “Hello” << std::endl;
}
```

using declaration

Explicit::Qualification

- A fully scope qualified name

iostream

```
namespace std
{
    class ostream { ... };
    ostream & endl(ostream &);
    extern ostream cout;
}
```

```
#include <iostream>

void eg()
{
   std::cout << “Hello” << std::endl;
}
```

Headers are meant to be included

- Using directives/declarations in a header will have an unknown span of effect - which entirely defeats the purpose of namespaces!
- In a header file use *only* explicit qualification

header.hpp

```
#include <string>

using namespace std;

string bad();
```

header.hpp

```
#include <string>

std::string good();
```

A Puzzle...

- (a << b) is syntactic sugar for operator<<(a, b)

```
std::operator<<(std::cout, “Hello\n”);
```

But we *<u>aren't</u>* required to qualify operator<< with std:: ?

```
#include <iostream>

void eg()
{
    std::cout << “Hello\n”;
}
```

Argument Dependent Lookup

- The compiler can look for the function in the namespaces of its arguments
- ADL - also known as Koenig lookup

```
namespace std
{

    ... operator<<(std::ostream &, const char *);
}
```

```
std::cout << “Hello, world\n”
```

Quiz

- Will this compile?
- If not, why not?

```
#ifndef GRAMMAR_LIB_GRAMMAR_HPP_INCLUDED
#define GRAMMAR_LIB_GRAMMAR_HPP_INCLUDED

namespace grammar_lib
{
    class grammar
    {
    public:
        //...
        void insert(non_terminal *);
    };
}

#endif
```

Answer

- No - the compiler does not magically somehow know an identifier is the name of a type

```
#ifndef GRAMMAR_LIB_GRAMMAR_HPP_INCLUDED
#define GRAMMAR_LIB_GRAMMAR_HPP_INCLUDED

namespace grammar_lib
{
    class grammar
    {
    public:
        //...
        void insert(non_terminal *);
    };
}

#endif
```

One Solution

- Given this header file...
- ...add a #include

non_terminal.hpp

```
namespace grammar_lib
{
    class non_terminal
    {
    public:
        //...
    };
}
```

```
#include "non_terminal.hpp"

namespace grammar_lib
{
    class grammar
    {
    public:
        ...
        void insert(non_terminal *);
    };
}
```

Another Solution

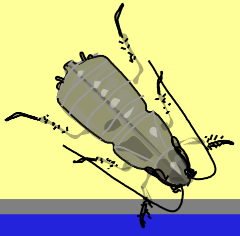
- Use a *forward declaration* - tell the compiler *only* that the identifier is the name of a class
- Reduces/exposes include dependencies :-)

```
#include ...

namespace grammar_lib
{
    class non_terminal;

    class grammar
    {
    public:
        ...
        void insert(non_terminal *);
    };
}
```

However...

- What you promise in a forward declaration has to *exactly* match the definition

```
namespace std
{
    class string;
}

namespace grammar_lib
{
    class grammar
    {
    public:
        grammar(const std::string & name);
        ...
    };
}
```



And sometimes...it doesn't

- You can't forward declare string
  - You have to #include <string>
- You can't forward declare stream classes
  - You can #include <iosfwd> though

```
namespace std
{
    template<typename CharType, ...>
    class basic_string
    {
        ...
    };

    typedef basic_string<char,...> string;
}
```

Quiz - 9 cases

- Which need a #include “wibble.hpp” and which only need a forward declaration of wibble?

```
class nine_cases
{
    void   one(wibble  );
    void   two(wibble *);
    void three(wibble &);

    wibble   four();
    wibble * five();
    wibble & six();

    wibble   seven;
    wibble * eight;
    wibble & nine;
};
```

Please discuss in your groups

Answer

- Only case seven requires a #include!!

  A forward declaration is sufficient for all others

```
class nine_cases
{
    void   one(wibble  );
    void   two(wibble *);
    void three(wibble &);

    wibble   four();
    wibble * five();
    wibble & six();

    wibble   seven;
    wibble * eight;
    wibble & nine;
};
```

one-six are declarations
not definitions

Self-Contained Header Files

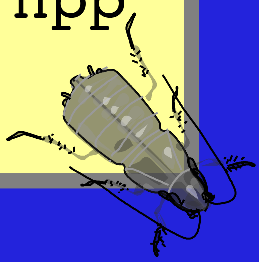
- To include one header you should *never* need to include another header
- Make the first #include in each source file its *own* header

wibble.hpp

```
class fubar;

class wibble
{
public:
    void (fubar *);
    ...
};
```

wibble.cpp

```
#include "fubar.hpp"
#include "wibble.hpp"
...
```



wibble.cpp

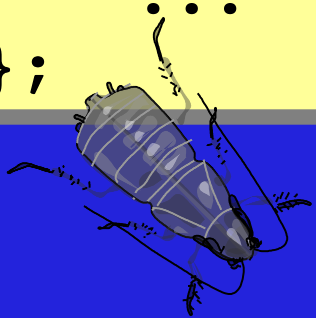
```
#include "wibble.hpp"
#include "fubar.hpp"
...
```

Unself-Contained Header Files

- To include some header files you need to include other header files first... :-(

wibble.hpp

```
class wibble
{
public:
    void (fubar *);
    ...
};
```



this header file does not forward declare fubar or #include fubar.hpp

...

but the problem is avoided like this...

```
#include "fubar.hpp"
#include "wibble.hpp"
...
```

source.cpp

Design is about being...

- Easy to use
  - clean abstractions that hide unimportant details
  - tests are examples of use - they shepherd design
- Easy to test
  - if you don't write tests you *will* end up with software that is hard to test - and that is *not* surprising
  - testability is a *key* criteria of design
- Easy to maintain
  - you are constantly battling against entropy
  - tests act as a safety net and encourage refactoring

Testing

- System test
  - all external dependencies in place
- Integration test
  - some external dependencies in place
  - some external dependencies mocked out
- Unit-test
  - all external dependencies mocked out
  - *Reliability* - no false positive/negative passes/fails
  - *Speed* - running unit-tests becomes the driver of how you program

Header File Summary

- Mirror the folder/namespace and file/class names in the macro guards
- Always use explicit qualification
- Use forward declarations when you can
- Use #include's only when you have to
- Ensure every header is self-contained; compilable in its own right
- Header files and tests represent the design
  source files are somewhat incidental!
  the tests tell us if they work
  and we don't have a choice about that!

C++ Foundation

Standard Libraries

Standard Libraries

- containers and iterators
- algorithms
- string
- iostream, stringstream
- pair
- functional
- the C library
- C++ in the future: boost, tr1, C++0x

Containers

- Sequential: vector, list, deque, queue, stack
- Associative: map, multimap, set, multiset

```
template<typename Type>
class list
{
public:
    bool empty() const;
    size_t size() const;

    void push_front(const Type &);
    void push_back(const Type &);
    void clear();
    ...
};
```

Iterators

- Modelled on pointers

```
template<typename Type>
class list<Type>
{
public:
    class iterator
    {
    public:
        Type & operator*() const;
        Type * operator->() const;
        iterator operator++();
        ...
    };
    bool operator==(iterator, iterator);
    bool operator!=(iterator, iterator);
};
```

Iterators

- A pair of iterators [begin, end) specifies a range

```
template<typename Type>
class list
{
public:
    ...
    template<typename It>
    list(It begin, It end);

    iterator begin();
    iterator end()
    ...
    void insert(iterator, const Type &);
    void erase(iterator);
    ...
};
```

Lots of <algorithm>s

sequence: non-modifying

adjacent_find, count, count_if, equal, *for_each*, *find*, *find_if*, find_end, find_first_of, mismatch, search, seach_n

sequence: modifying

copy, copy_backward, generate, generate_n, fill, fill_n, iter_swap, partition, replace, replace_if, replace_copy, replace_copy_if, *remove*, remove_if, remove_copy, remove_copy_if, reverse, reverse_copy, rotate, rotate_copy, random_shuffle, stable_partition, swap, swap_ranges, *transform*, *unique*, unique_copy

sorting

nth_element, partial_sort, partial_sort_copy, *sort*, stable_sort

Lots of <algorithm>s

binary search

```
binary_search, equal_range,
lower_bound, upper_bound
```

merge

```
inplace_merge, includes, merge, set_union,
set_intersection, set_difference,
set_symmetric_difference
```

heaps

```
make_heap, push_heap, pop_heap, sort_heap
```

min-max

lexicographic_compare,
min, *max*, min_element, max_element,
next_permutation, prev_permutation

Algorithms

- Function templates - iterator pairs

```
template<typename Iter, typename Type>
Iter find(Iter at, Iter end, const Type & value)
{
    for (; at != end; ++at)
        if (*at == value)
            break;
    return at;
}
```

```
template<typename Iter, typename Pred>
Iter find_if(Iter at, Iter end, Pred pred)
{
    for (; at != end; ++at)
        if (pred(*at))
            break;
    return at;
}
```

Algorithms

- Function templates - iterator pairs

```
template<typename InputIter,
         typename OutputIter,
         typename UnaryOp>
OutputIter transform(InputIter at, InputIter end,
                     OutputIter result,
                     UnaryFunc f)
{
    while(at != end)
    {
        *result = f(*at);
        ++result;
        ++at;
    }
    return result;
}
```

Writing loops?

- Many loops can be refactored to an algorithm

```
typedef std::list<int>::iterator iterator;
for (iterator at = values.begin();
     at != values.end();
     ++at)
{
    std::cout << *at << ',';
}
```

```
void couter(int value)
{
    std::cout << value << '.';
}
std::for_each(values.begin(), values.end(),
              couter);
```

string

- Goodbye char * horribleness

```
class string
{
public:
    string();
    string(const char *);

    size_t size() const;
    bool empty() const;
    void clear();
    char & operator[](size_t);
    const char & operator[](size_t) const;
    ...
};
```

simplified

string

- Retrofitted to STL container model

```
class string
{
public:
    class iterator;
    class const_iterator;

    template<typename It>
    string(It, It);

    iterator begin();
    iterator end();
    const_iterator begin() const;
    const_iterator end() const;
    ...
};
```

simplified

Streaming << or >>

- Write the stream object first, then the operator
  - >> to indicate data flowing out of the stream
  - << to indicate data flowing into the stream

```
void in(istream & is)
{
    int value;
    is >> value;
    ...
}
```

```
void out(ostream & os)
{
    int value = 42;
    os << value;
    ...
}
```

Streaming

- Providing operator<< allows you to write to files

```
ostream & operator<<(ostream &, const date &);
```

```
#include <fstream>

void eg()
{
    date xmas(2011,12,25);
    std::ofstream ofs(“date.txt”);
    ofs << xmas;
}
```

date.txt ←

```
2011/12/25
```

Streaming

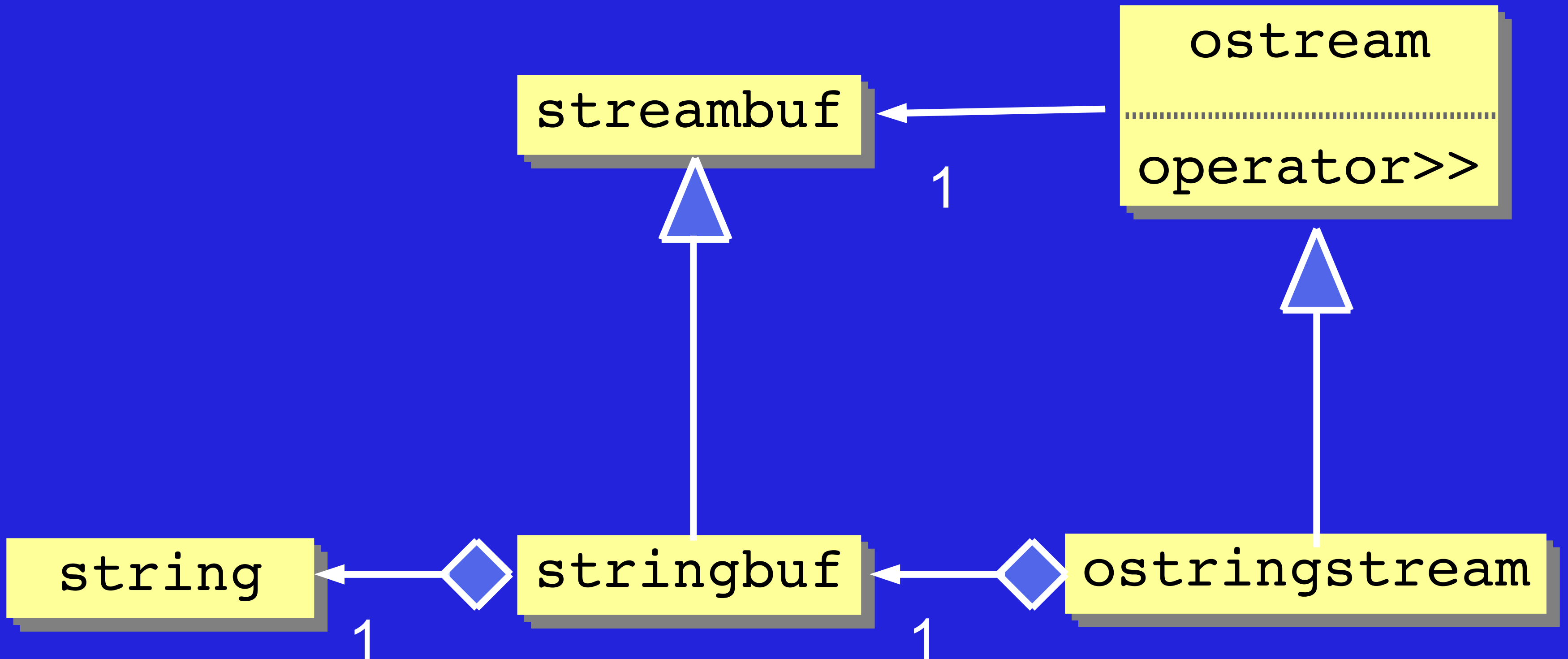
- Providing operator<< also allows you to write to strings - very handy for test diagnostics

```
ostream & operator<<(ostream &, const date &);
```

```
#include <sstream>

void eg()
{
   date xmas(2011,12,25);
   std::ostringstream oss;
   oss << xmas;
   assert(oss.str() == "2011/12/25");
}
```

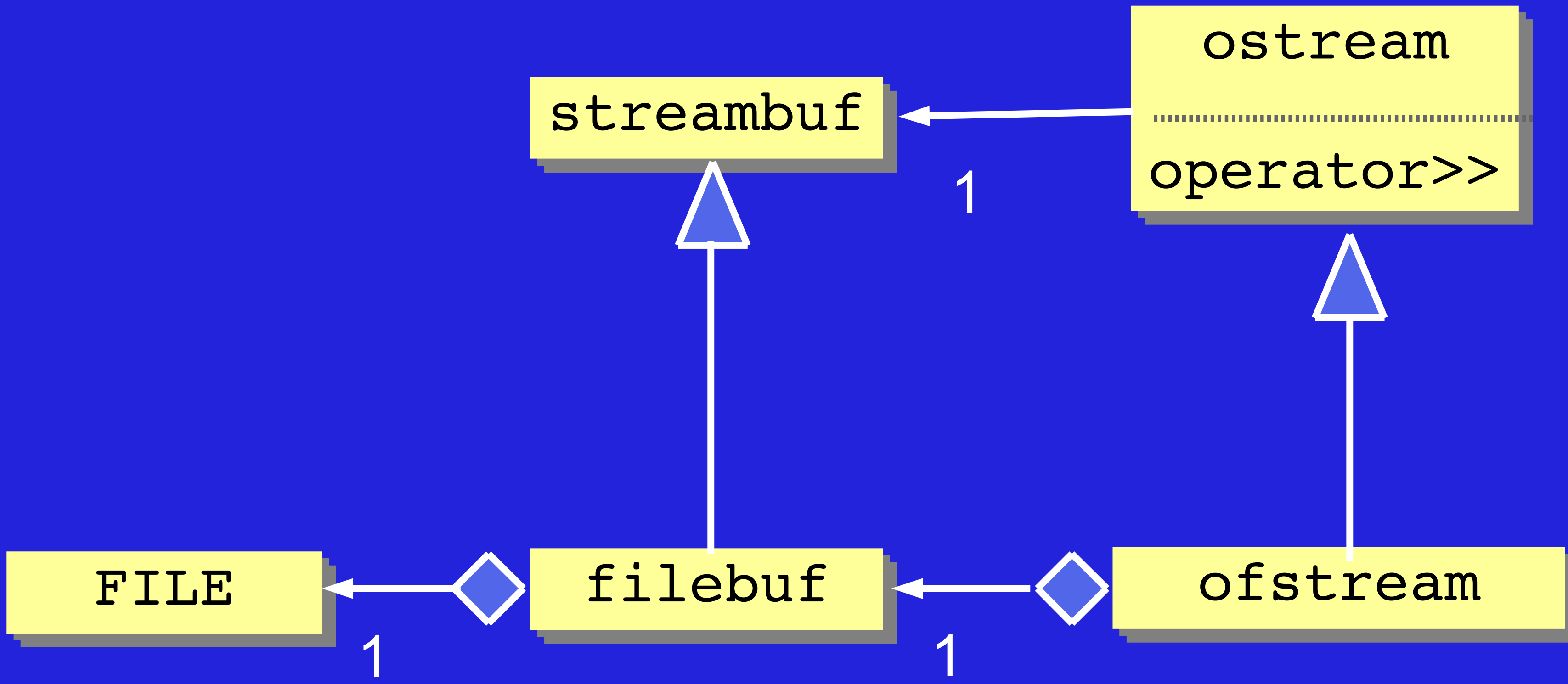
streambuf

- Buffers characters manipulated by a stream
- Subclassed in parallel with the stream



streambuf

- Buffers characters manipulated by a stream
- Subclassed in parallel with the stream



pair<T1,T2>

- A simple two-tuple in <utility>

```
template<typename T1, typename T2>
struct pair
{
    typedef T1 first_type;
    typedef T2 second_type;

    T1 first;
    T2 second;

    pair()
        : first(T1()), second(T2()) {}
    pair(const T1 & f, const T2 & s)
        : first(x), second(s) {}
    template<typename U, typename V>
    pair(const pair<U,V> & p)
        : first(p.first), second(p,second) {}
};
```

Often usable instead of a small struct

make_pair

- A simple helper function template

```
template<typename T1, typename T2>
pair<T1,T2> make_pair(T1 f, T2 s)
{
    return pair<T1,T2>(f, s);
}
```

```
std::pair(42, answer);
```

```
std::pair<int,std::string>(42, answer);
```

```
std::make_pair(42, answer);
```

<functional>

- Provides a framework and classes usable as predicates for algorithms and containers

```
void eg()
{
    int values[] = { 2,5,8,3,7 };

    std::sort(values, values + 5);
    // [2,3,5,7,8]

    std::sort(values, values + 5,
        std::greater<int>());
    // [8,7,5,3,2]
}
```

<functional>

- Provides a framework and classes usable as predicates for algorithms and containers

```
template<typename T> struct equal_to;      // ==
template<typename T> struct not_equal_to;  // !=
template<typename T> struct less;          // <
template<typename T> struct less_equal;    // >=
template<typename T> struct greater;       // >
template<typename T> struct greater_equal; // >=
```

```
template<typename T>
struct greater : ...
{
    bool operator()(const T & x, const T & y) const
    {
        return x > y;
    }
};
```

The C Library

- Most C <*header*.h>'s have a corresponding std namespace wrapping C++ <c*header*>

```
#include <string.h>

struct c_str_less
{
    bool operator()(const char * lhs, const char * rhs) const
    {
        return strcmp(lhs, rhs) < 0;
    }
};
```

```
#include <cstring>

struct c_str_less
{
    bool operator()(const char * lhs, const char * rhs) const
    {
        return std::strcmp(lhs, rhs) < 0;
    }
};
```

✓

http://www.boost.org

- Where future C++ libraries are born and grow
  - Aims to establish reference implementations of existing practice
  - High quality
  - Peer reviewed
  - Proving ground for TR1 and TR2
  - Any, Threading, Date and Time, Lambda, FileSystem, Parsing, Serialization, Tokenization, Graphs, Hashing

Technical Report 1 (tr1)

- Library components slated for C++0x (sic)
  - <memory> shared_ptr<T>, weak_ptr<T>
  - <functional> function<T> - polymorphic function call
  - <type_traits> meta-programming utilities
  - <random> number generators
  - <tuple>
  - <array> fixed size array
  - <unordered_set> - hash based set
  - <unordered_map> - hash based map
  - <regex>

www.open-std.org/jtc1/sc22/wg21/docs/papers/2005/n1745.pdf

C++ Foundation

Object Oriented Programming

Object Oriented Programming

- encapsulation
- information hiding
- liskov substitution principle
- parameterize from above
- single repsonsibility principle
- separation of concerns
- dependency
- abstraction
- patterns

En(capsule)ation

- Data and functions can be bundled together

```
struct file
{   ...
};
int getc(file*);
int ungetc(int, file*);
```

```
class file
{   ...
    int getc();
    int ungetc(int);
};
```

- An access restriction mechanism

```
class file
{
public:
    int getc();
    int ungetc(int);
private:
    ...
};
```

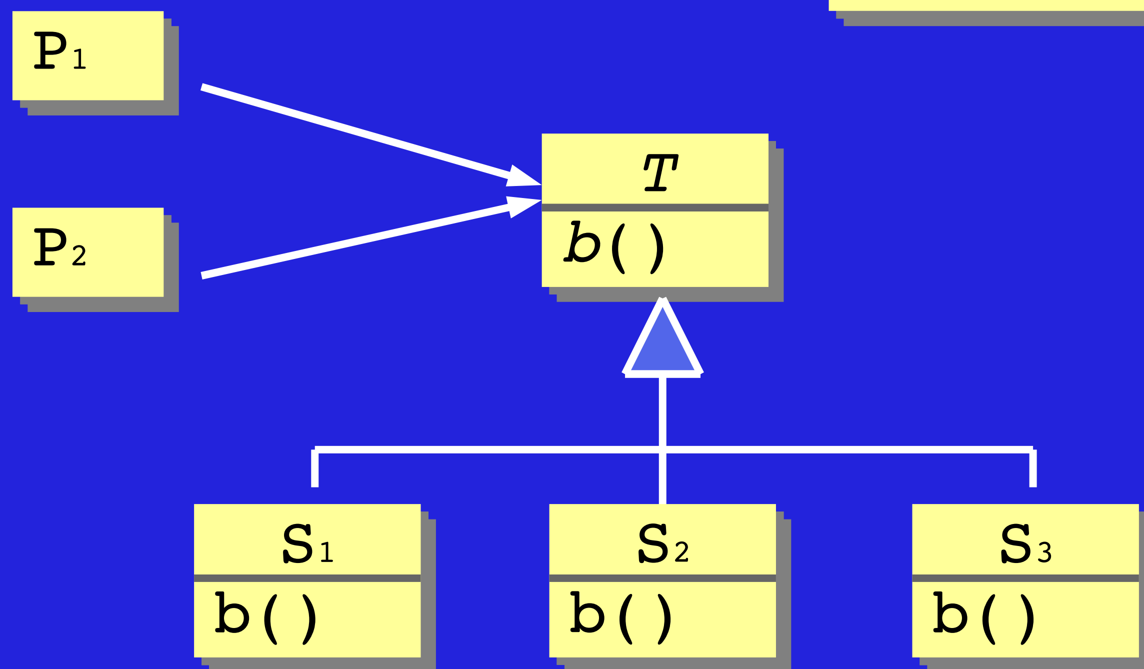
Information Hiding in C++

- We hide information partly so we can change what's hidden and limit the change's impact
  - public - private
    - change requires recompilation
  - header file - source file
    - change of implementation requires relinking
  - opaque types
    - change of representation requires relinking
  - inheritance hierarchies
    - change of type does not require relinking!

Liskov Subtitution Principle

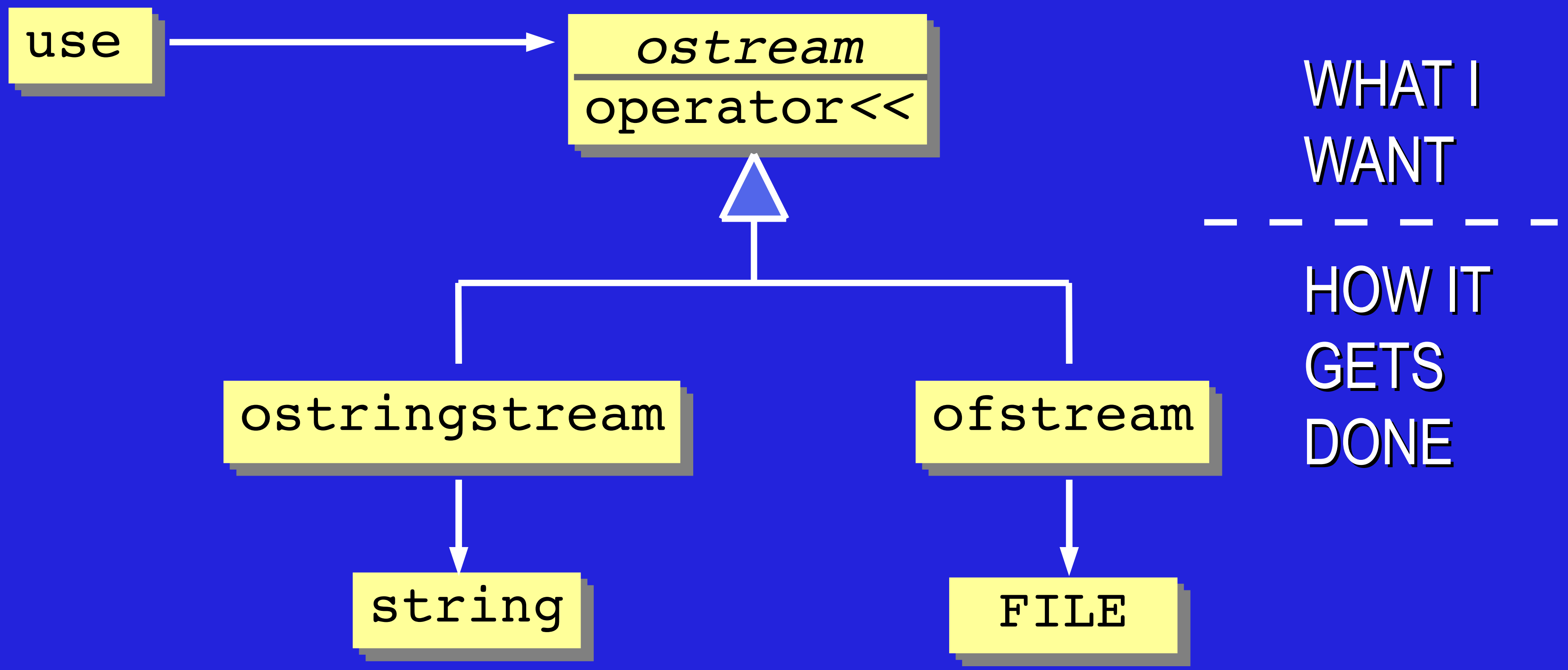
If for each object o1 of type S there is an object o2 of type T such that for all programs P defined in terms of T, the behaviour of P is unchanged when o1 is substituted for o2, then S is a subtype of T.

Barbara Liskov



Liskov Subtitution Principle

- For example...



Hard-wired From Below :-(

- Complicates testing, increases dependencies

```
struct date
{   ...
    void print() const
    {
        std::cout << ...;
    }
};
```

Non-parameterized
Fixed below / inside

```
void example(date when)
{
    when.print();
}
```

Not-parameterized
from above / outside

Parameterize From Above :-)

- Aim to make parameterization an explicit and visible part of the public api of a class/method

```
struct date
{   ...
    void print(std::ostream & os) const
    {
        std::cout << ...;
    }
};
```

```
void example(date when)
{
    when.print(std::cout);
}
```

Parameterized
from above / outside

Single Responsibility Principle

- A class should be responsible for one thing and one thing only

```
struct date
{   ...
    void print(std::ostream & os) const
    {
        std::cout << ...;
    }
};
```

2011/12/25

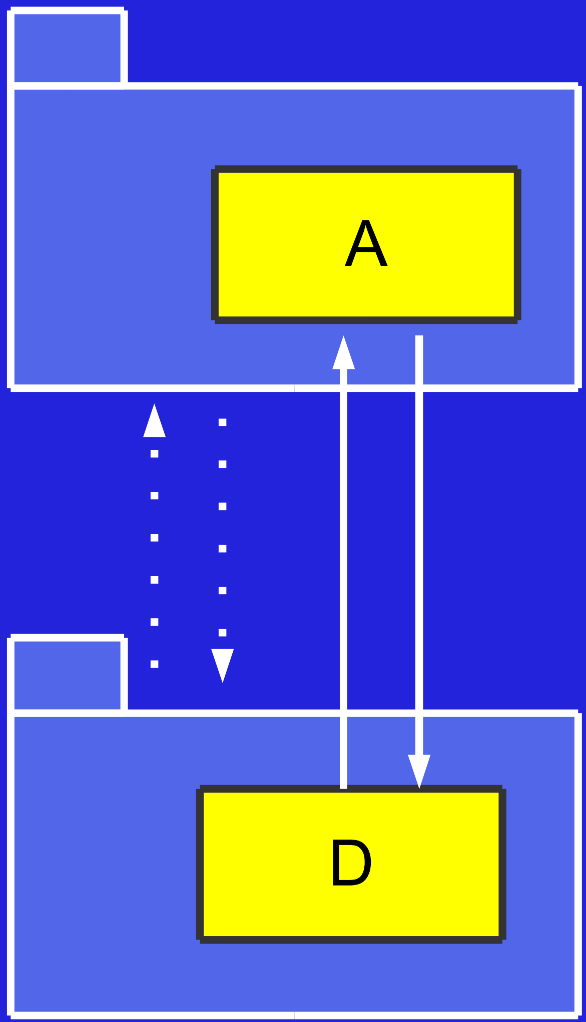
?

```
struct date
{   ...
    int year() const;
    int month() const;
    int day() cont;
};
```

Separation of Concerns

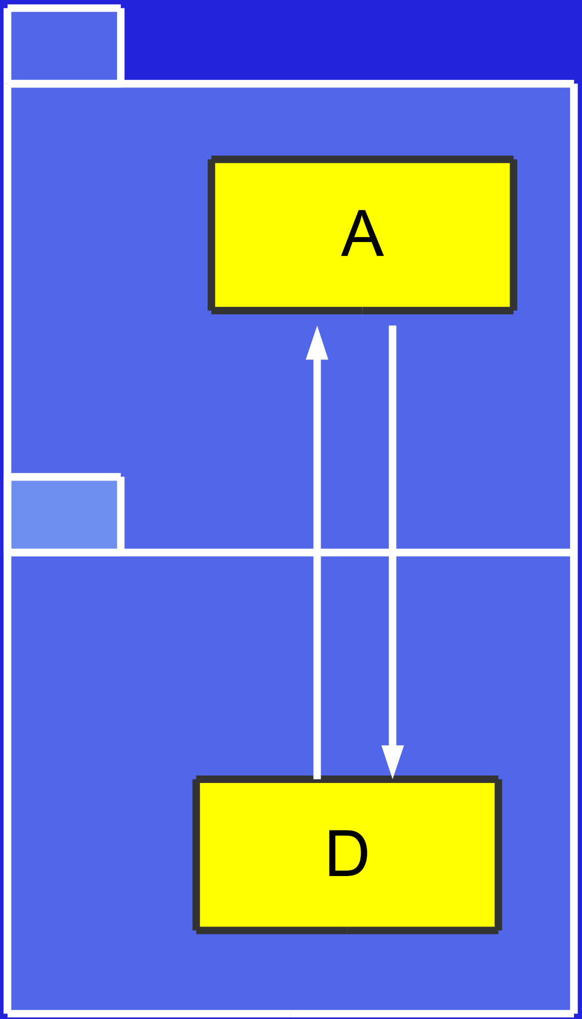
- Codebases tend to have a lot of effort focused on its functionalilty and not so much effort focused on its structure :-(
- Separate but dependent :-)

Dependency



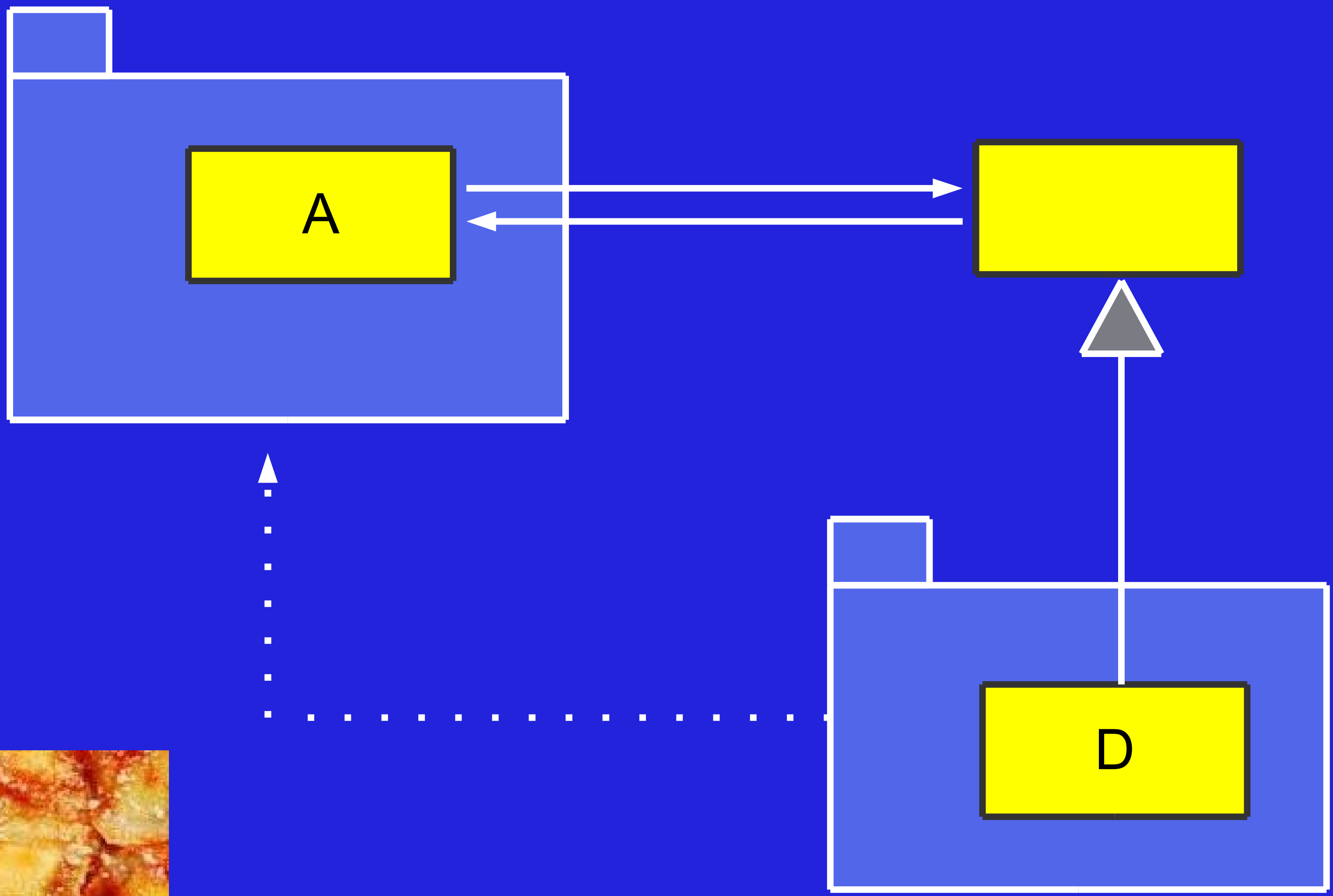
Claimed

Dependency

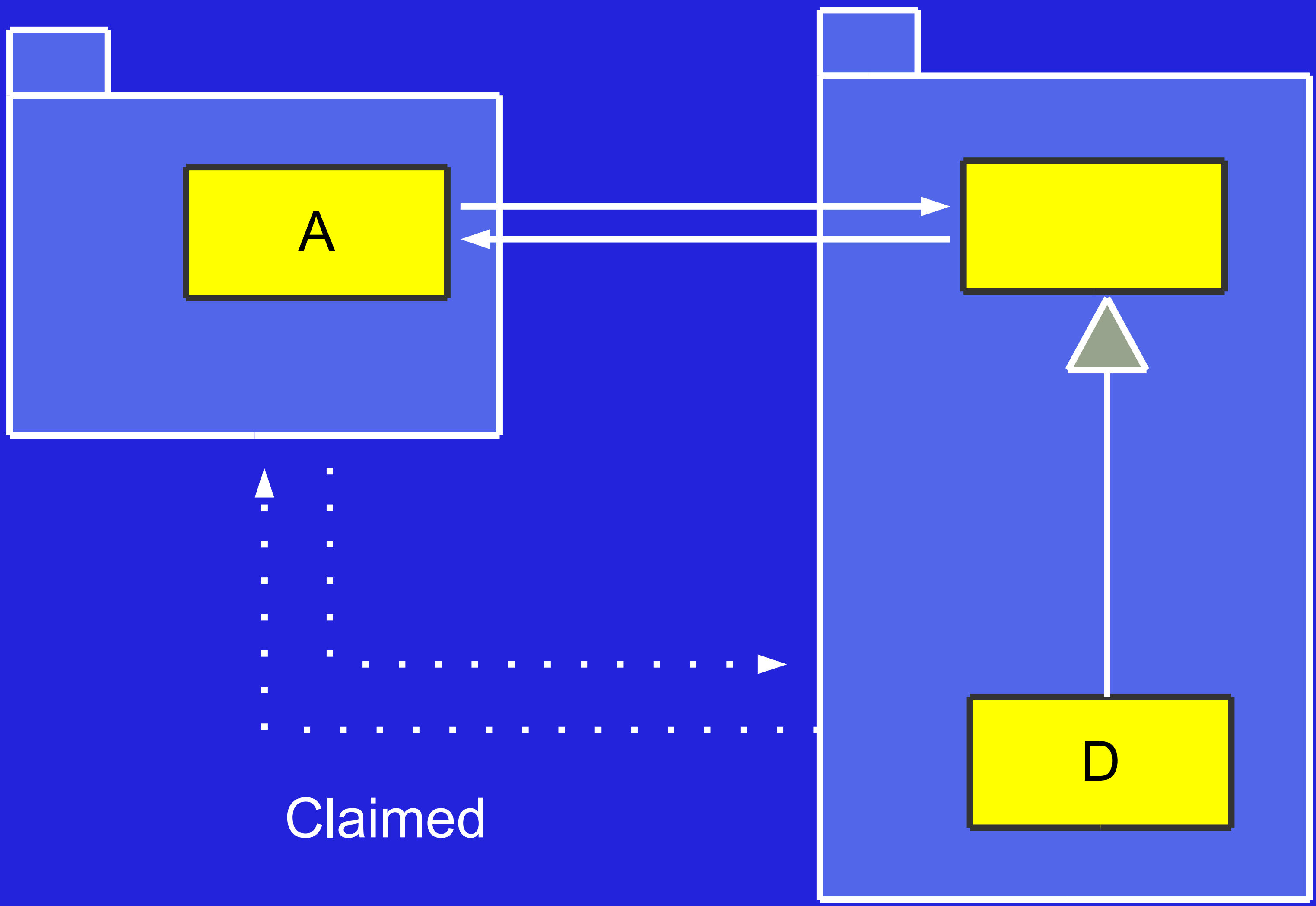


Actual

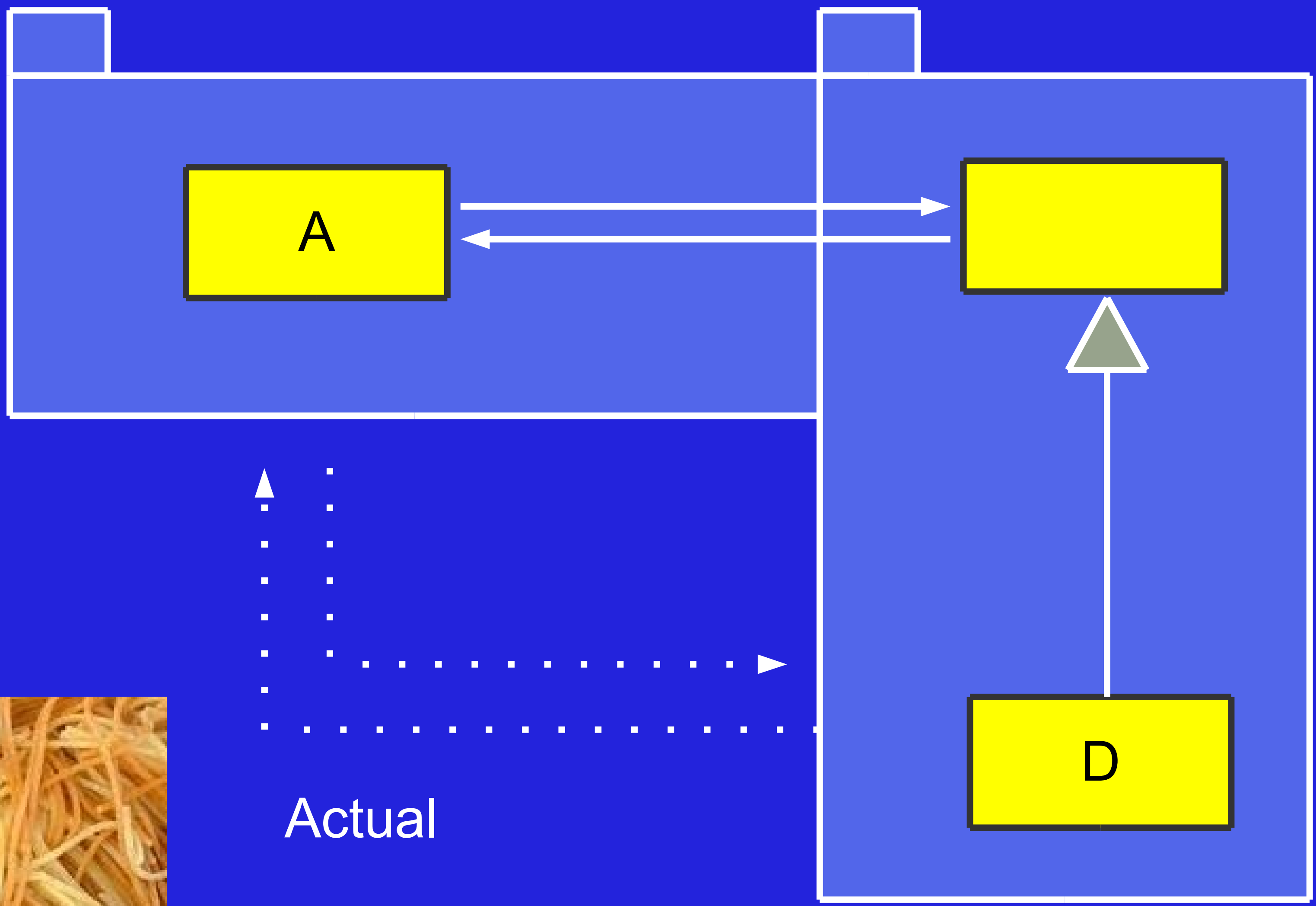
Dependency



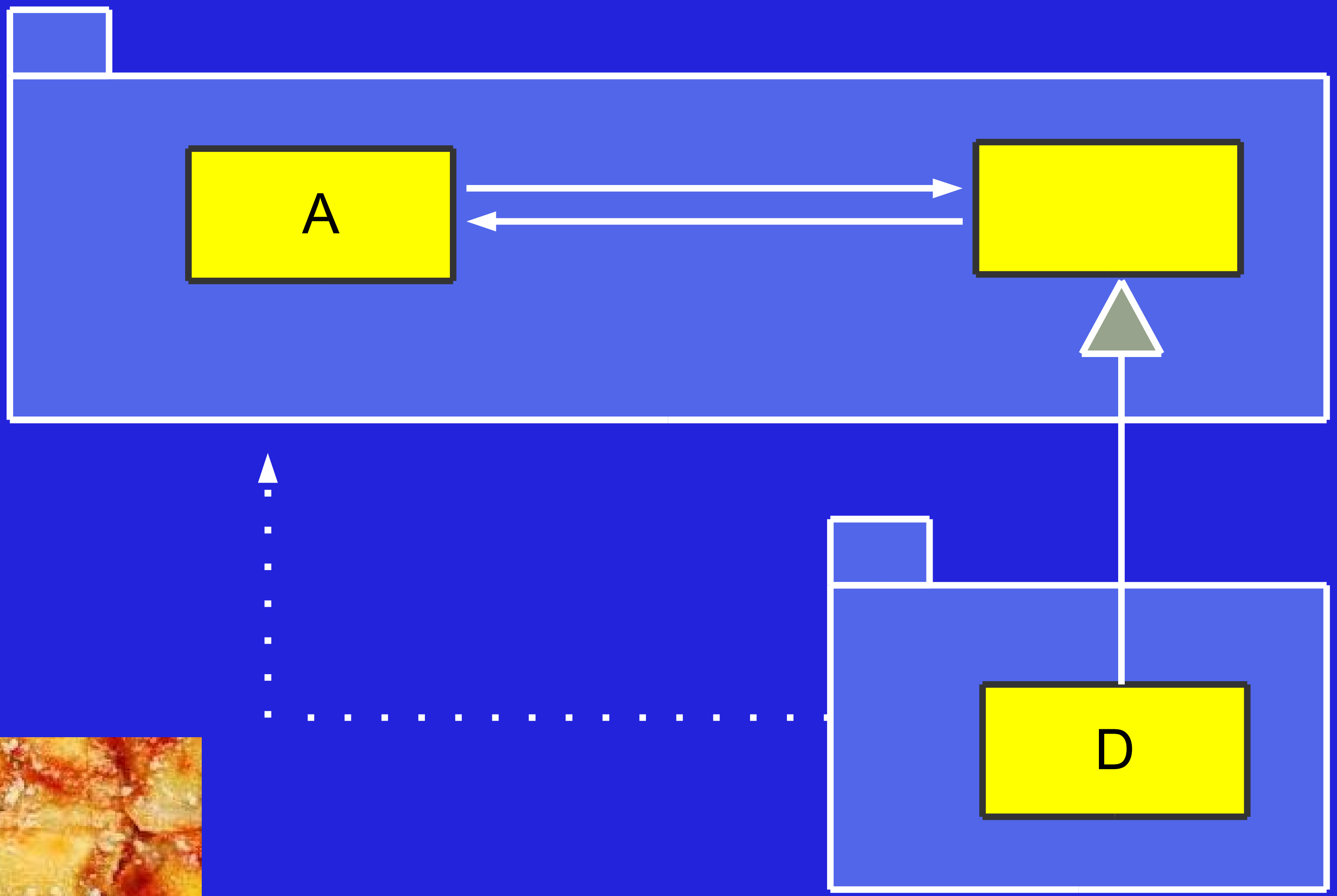
Dependency



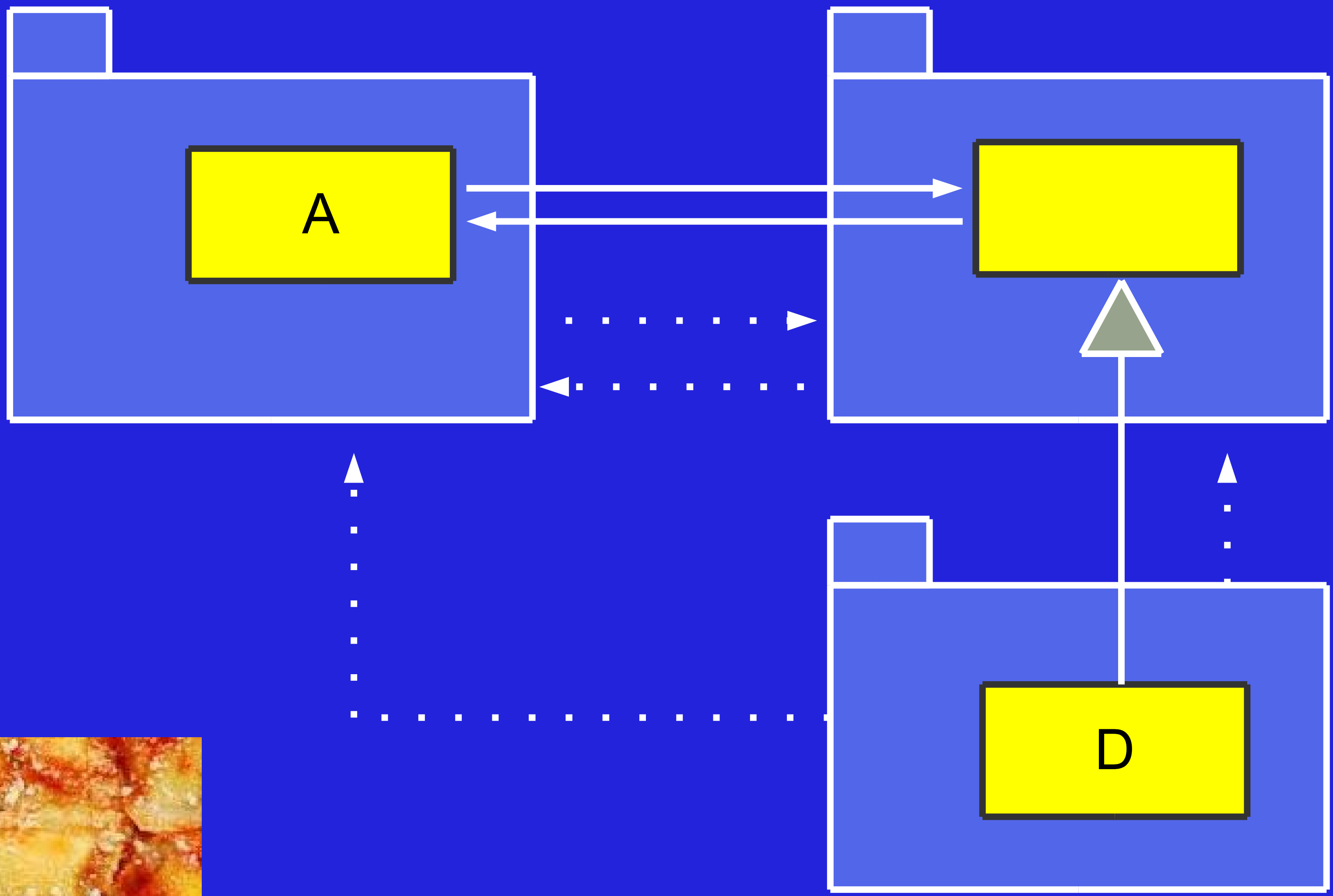
Dependency



Dependency



Dependency

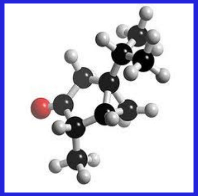


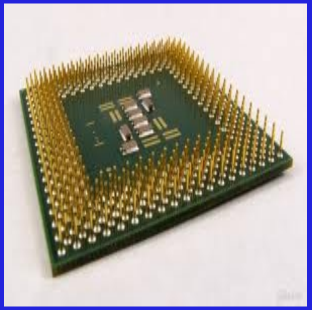
Abstraction

- We also hide information when creating crisp new semantic levels

Being abstract is something profoundly different from being vague... The purpose of an abstraction is not to be vague, but to create a *new semantic level* in which one can be absolutely precise.

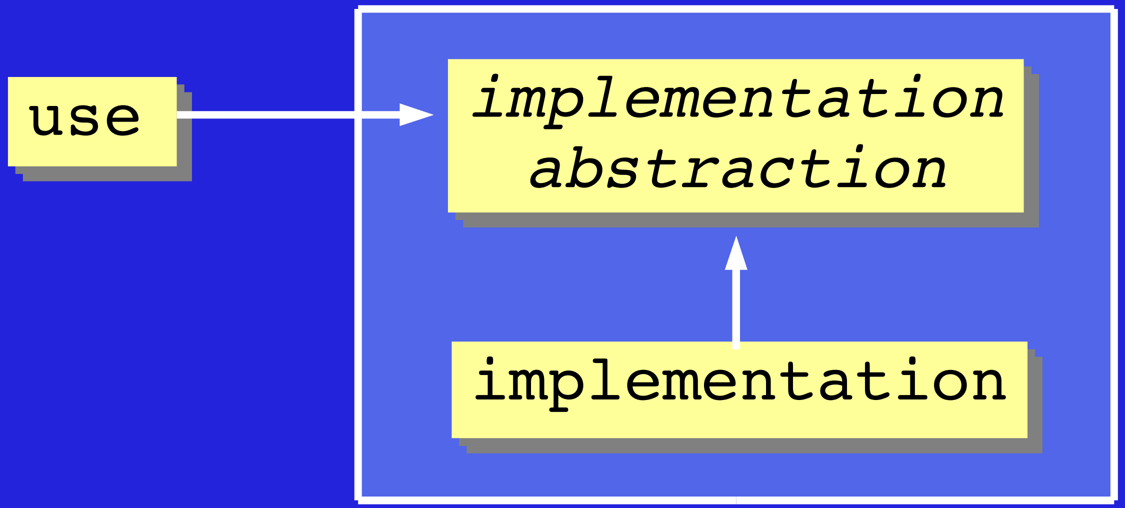
Edsger Dijkstra



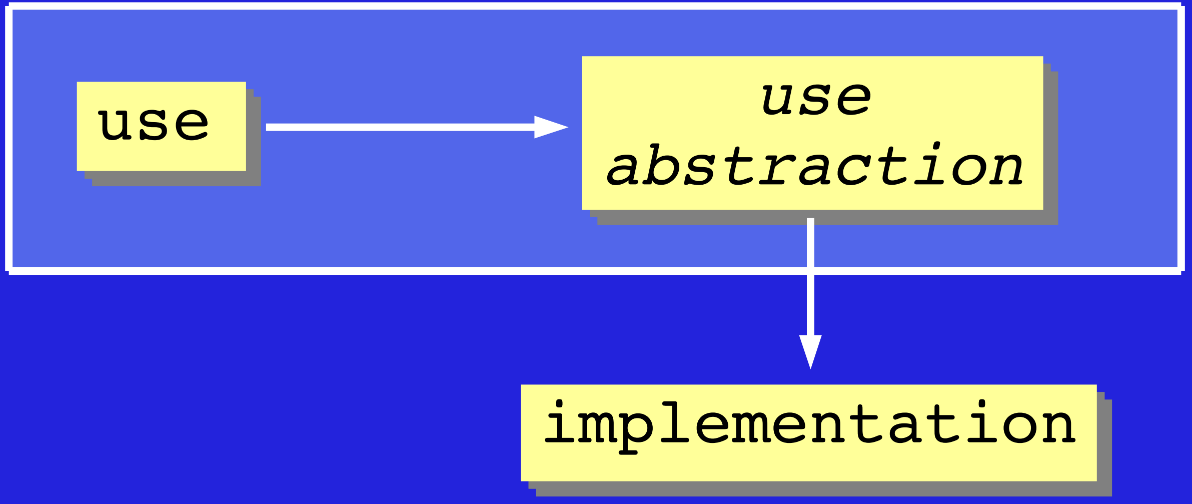


Abstraction

- How different are the semantic levels?



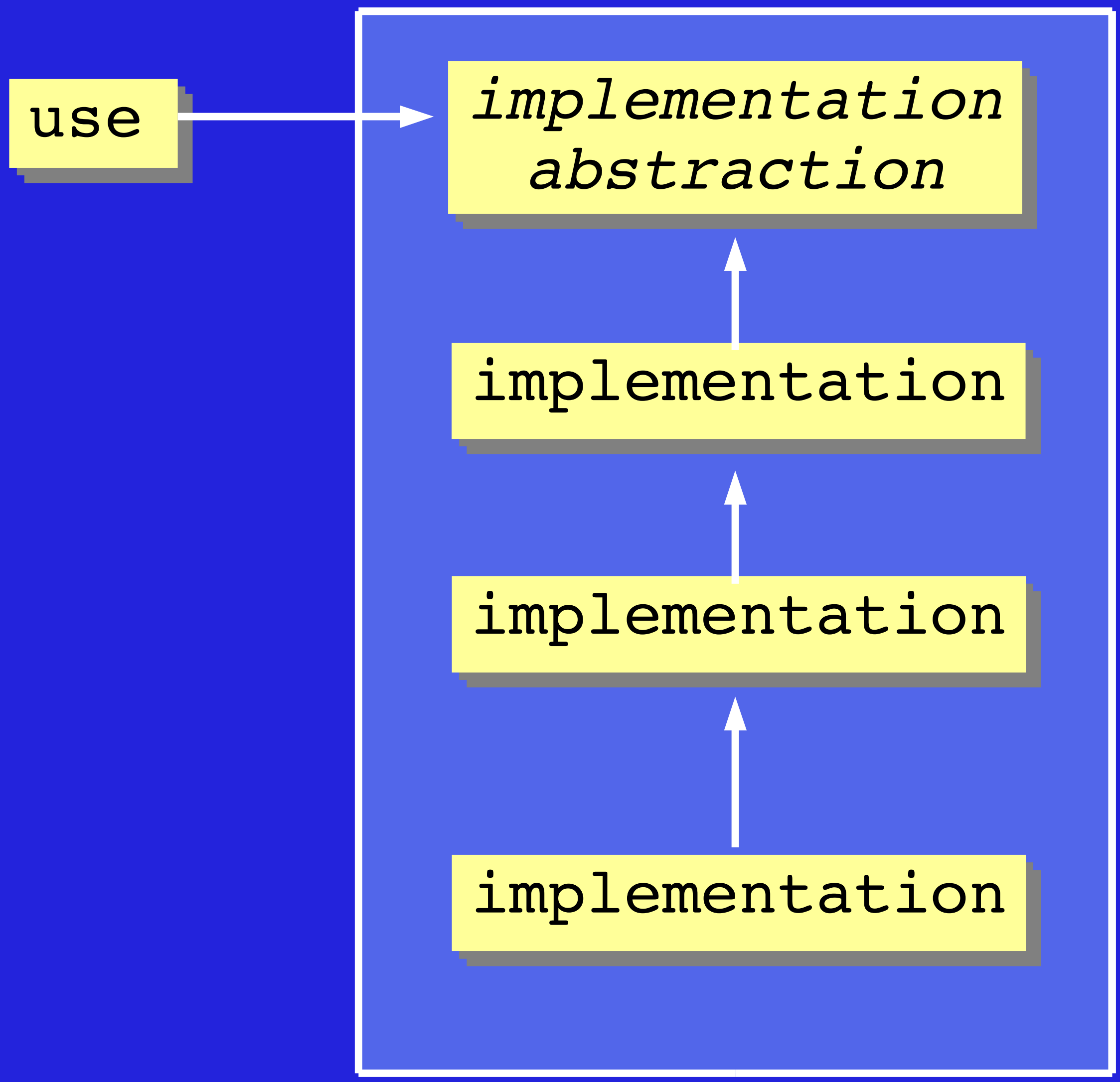
Some abstractions are weak and abstract away very little



Some abstractions are strong and abstract away a lot more

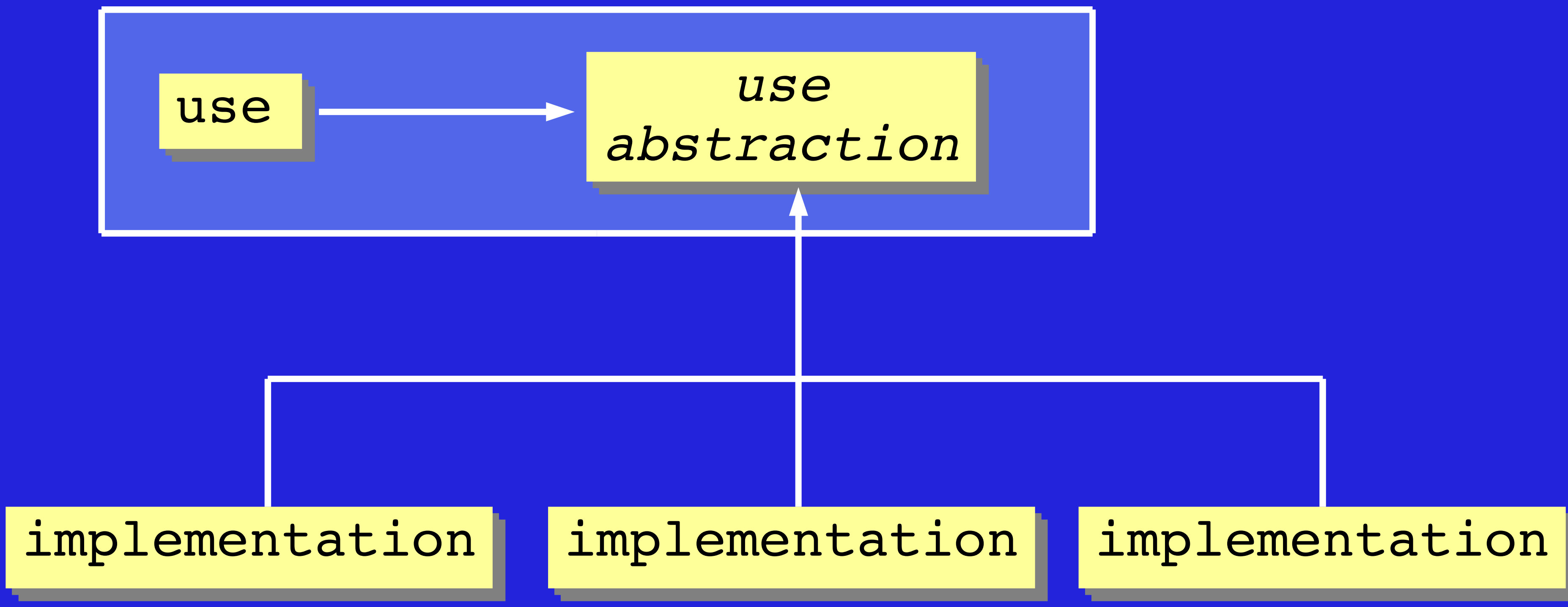
Abstraction?

- How much focus is on the implementation?



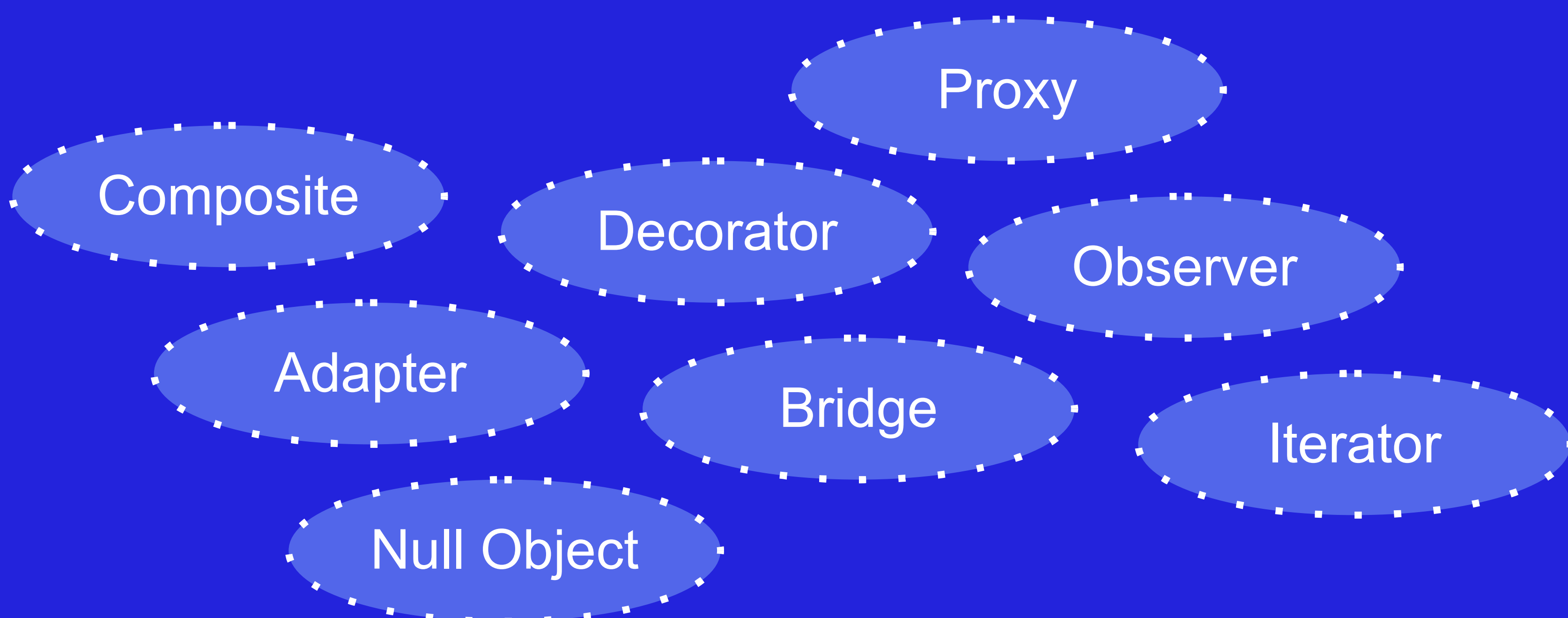
Abstraction

- How much focus is on the structure?



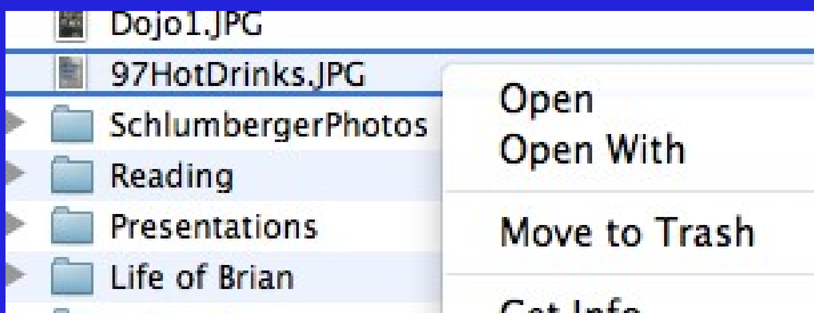
Patterns

- A class is not a useful unit of design!
- Patterns help you raise the level of abstraction
- Patterns document the role each class plays in a cluster of collaboration
- There are many named patterns

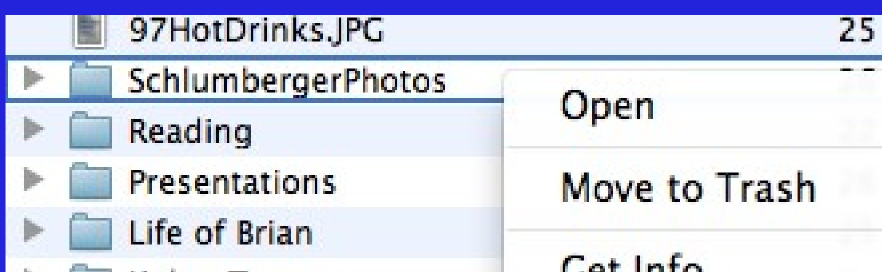


Deleting Files & Folders on a Mac

- Right click on the file or folder
- Click Move To Trash

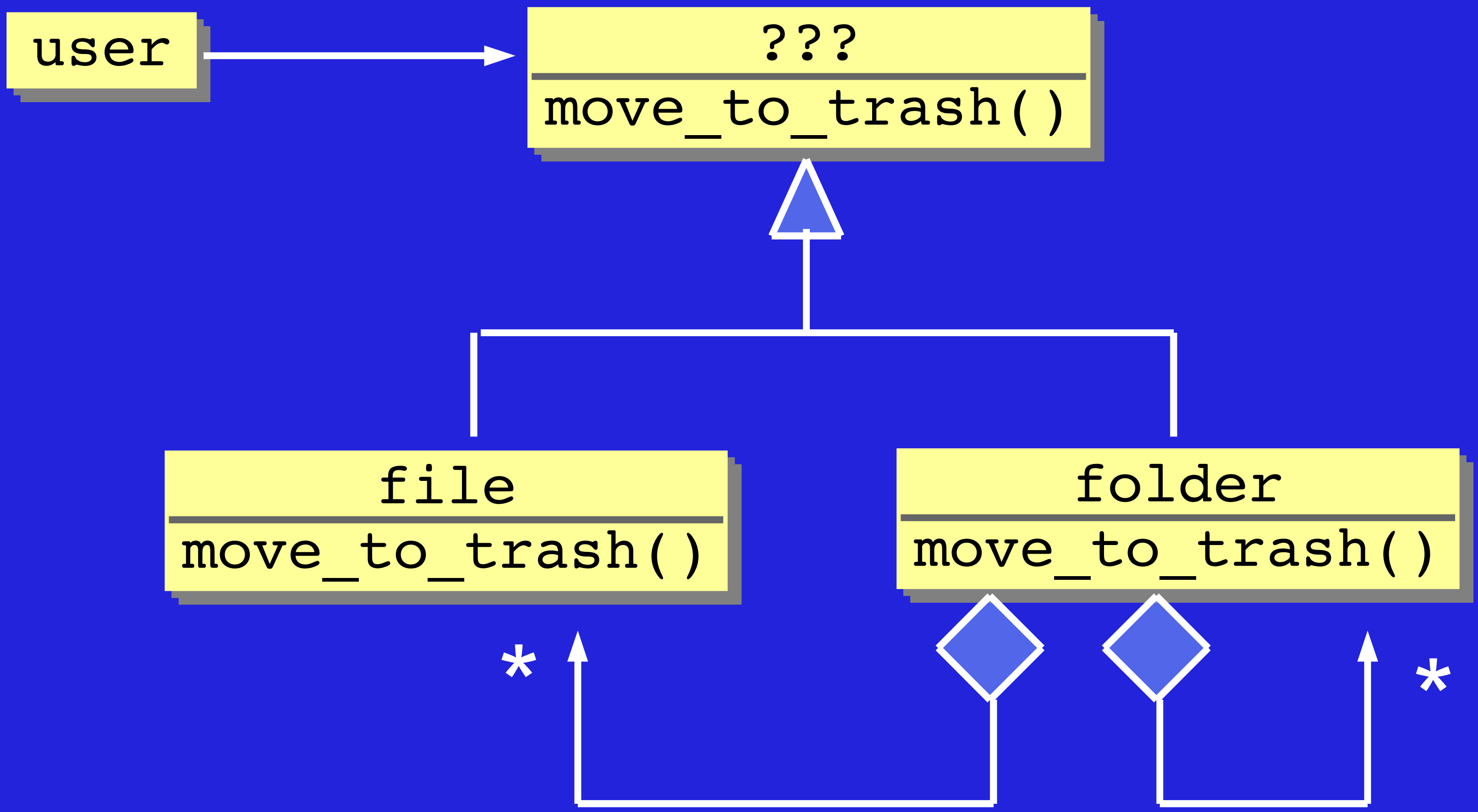


file
move_to_trash()

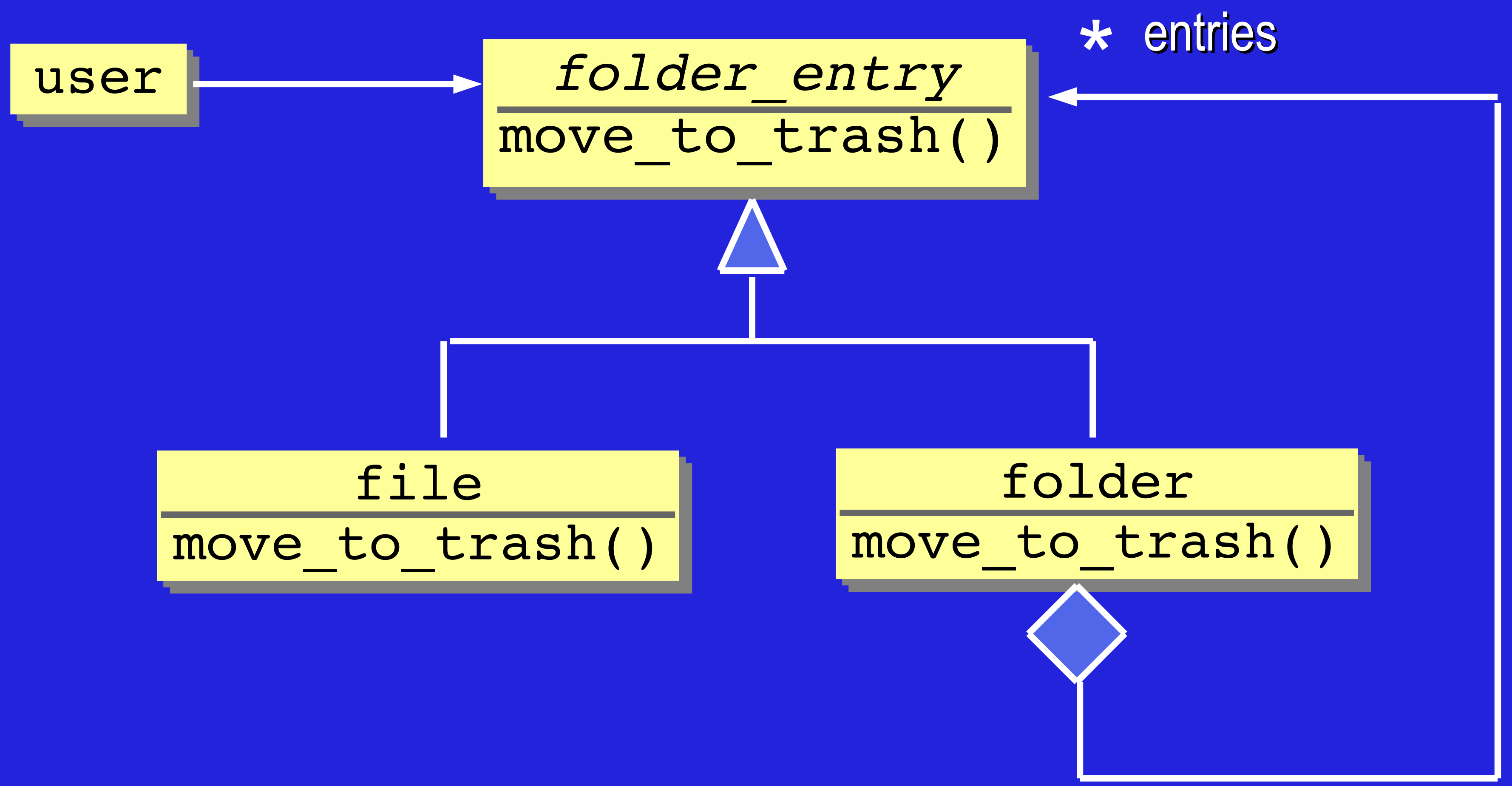


folder
move_to_trash()

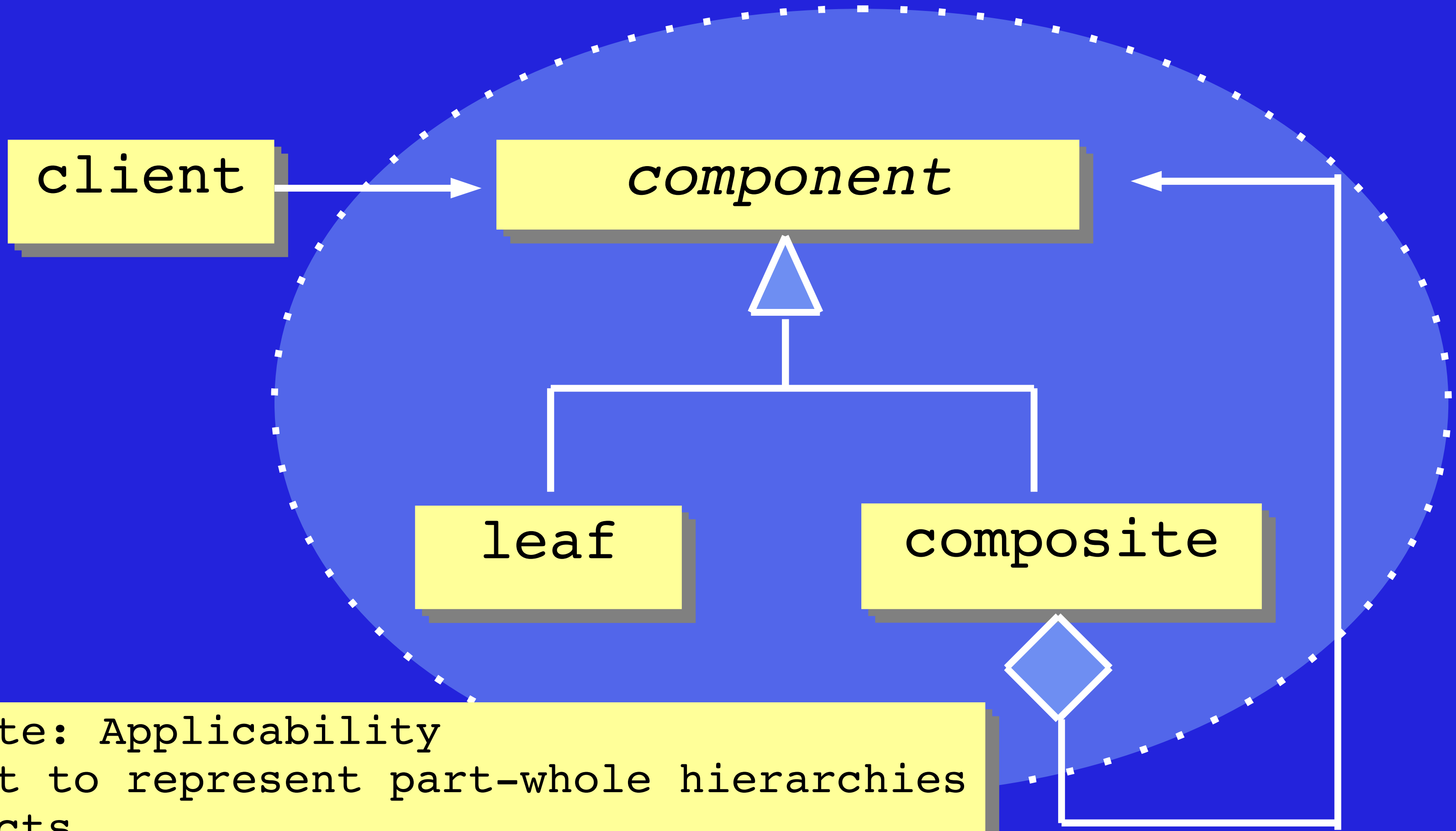
Files and Folders



Files and Folders and Folder Entries

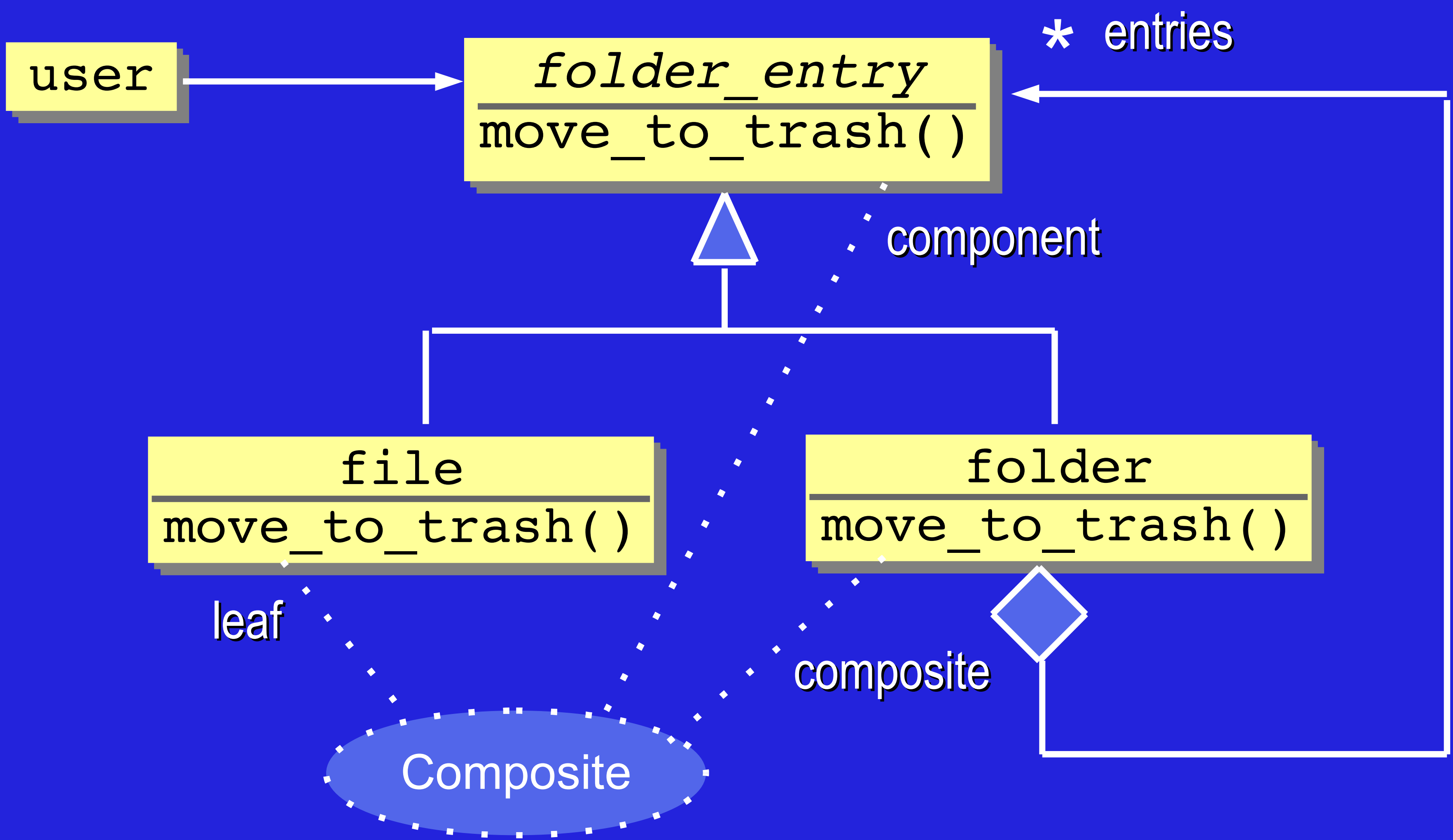


The Composite Pattern

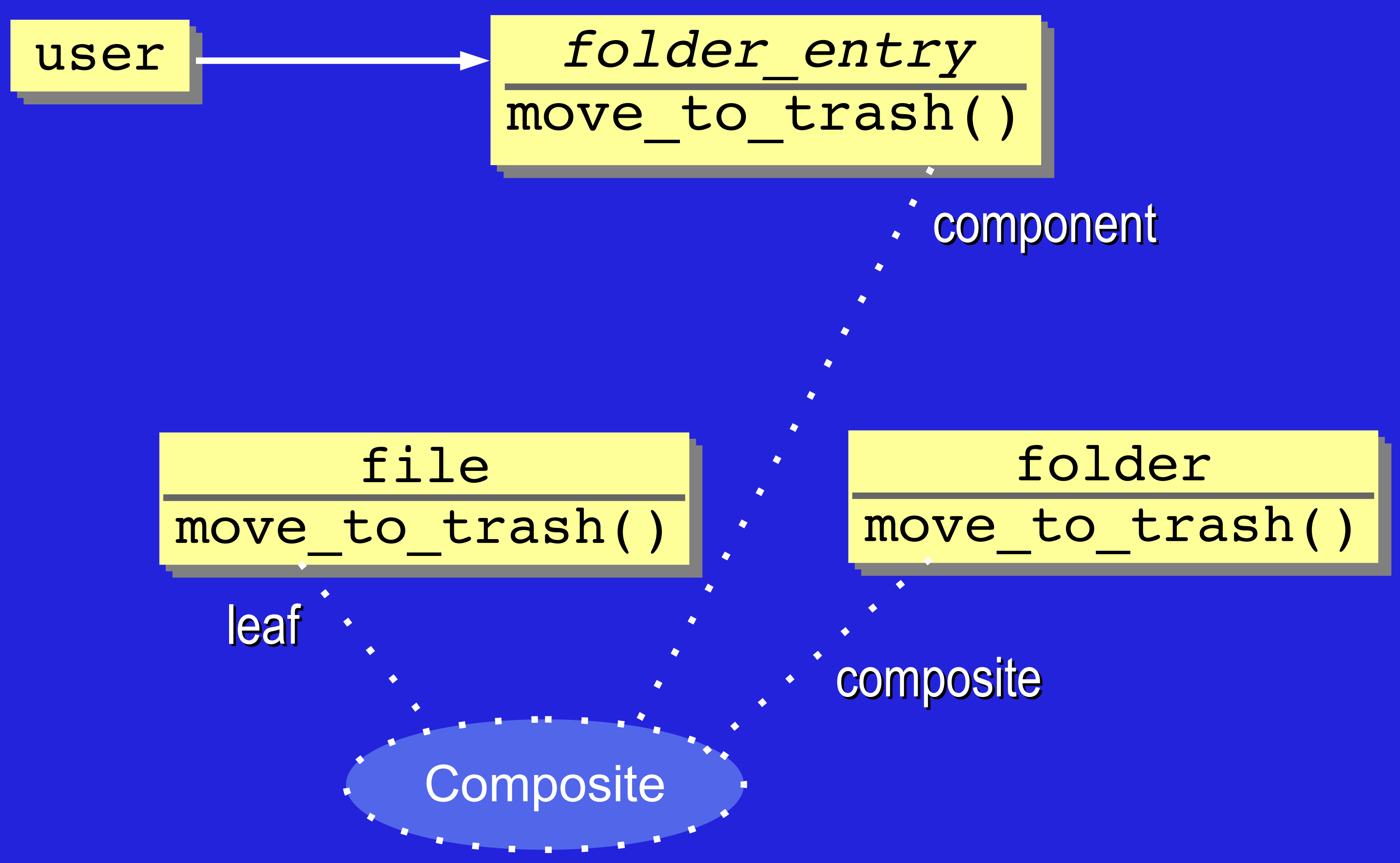


Composite: Applicability
You want to represent part-whole hierarchies of objects.
You want clients to be able to ignore the differences between compositions of objects and individual objects. Clients will treat all objects in the composite structure uniformly.

Files and Folders



Patterns


user
folder_entry
move_to_trash()
component
file
move_to_trash()
folder
move_to_trash()
leaf
composite
Composite