CODING FOR KEEPS

How to Write a Function

Chuck Allison

Better Software, June 2011

(slides available at freshsources.com/bs)

Functions

- Are the *verbs* of computing
  - The “life” of objects
- They encapsulate *algorithms*
- Are a low-level currency for *reuse*
  - the “pocket change”
  - Classes are “dollars”
  - Modules are “mutual funds” :-)

A Well-Designed Function is…

- Easier to *implement*
- Easier to *test*
- Easier to *read*
- Easier to *maintain*
- Easier to *reuse*
- A key component of *quality software*

Objectives for Today

- Appreciate the *principles* of good function design
  - writing *cohesive* functions
  - programming at a *higher level*

- Be familiar with the different kinds of *function architectures* available in modern programing languages
  - anonymous functions
  - delegates (aka “closures”)
  - generators and coroutines
  - pure functions

Agenda

- Fundamentals of Function Design
- Functions as First-Class Objects
- Polymorphic Functions
- Resource-Friendly Functions
- Parameter Passing Techniques (as time allows)

Along the Way We Will See…

- Higher-order and Anonymous Functions
- Nested functions, Closures and Delegates
- Generators and Coroutines
- Contract Programming
- Functions as Transactions
- Examples will be in C++ (2011), Python, and D (2.x)

What We've Heard

"Solutions should be as simple as possible, but no simpler"

-- Albert Einstein?

What Einstein Really Said

“It can scarcely be denied that the supreme goal of all theory is to make the irreducible basic elements as simple and as few as possible without having to surrender the adequate representation of a single datum of experience.”

- *Philosophy of Science*, Vol. 1, No. 2 (April 1934), p. 165.

What We Should Also Remember

"*Functions* should be as simple as possible, but no simpler"

FUNCTION DESIGN

Fundamentals

Fundamentals of Function Design

- A function should be as *self-contained* as possible
  - it does its *one job* but no more; no extraneous code
- Black-box Model of a Function:

- *Dependencies* on external objects should be *minimal*
  - Preferably only through *parameters* and *return values*

Functions vs. Procedures

- Side Effects!
  - changes other than in return values
- Procedures change *non-local* state
  - behind the caller's back
- We routinely do this with objects and methods
  - Complex objects are hard to debug
  - Getters and Setters are usually Problematic

Suspicious Types of Coupling

- Access to *variables in other scopes*
  - *global* variables
  - other *non-local* variables, e.g. via *reference parameters*
- Access to *functions* inside other scopes
- Access to *types* inside other scopes
- All these dependencies can *complicate* software

Case Study: String Tokenizing

- Consider C's **strtok** function

- `char* strtok(char* search, const char* break);`

- The string, **search**, is traversed and *modified*
  - characters in **break** are skipped
  - a '\0' (NUL) replaces the first break character after the token
  - it *remembers* where it left off for subsequent calls
  - the position of the first non-break character is returned

- Note: **strtok** must be called in *2 different ways*

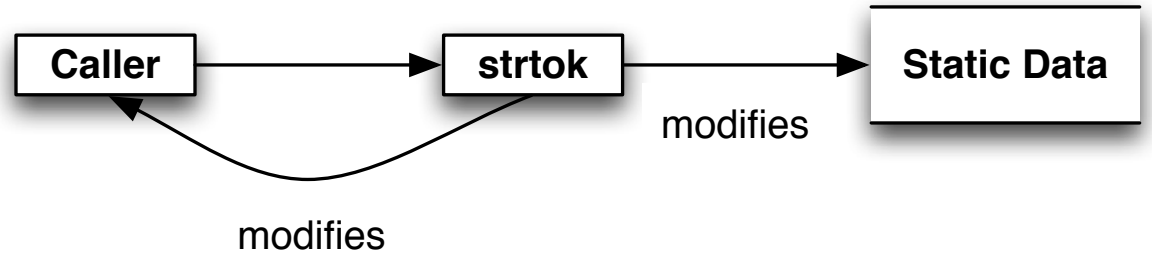
strtok Example

```
int main() {
    char search[BUFSIZ];
    strcpy(search,"This is 1just2a3test#.");
    char brkset[] = " \t\n\r\f\v`~!@#$%^&*()-_=+;:'\",<.>/”
                    "?01234567890";
    char* tokenptr = strtok(search,brkset);
    while (tokenptr != 0) {
        cout << tokenptr << endl;
        tokenptr = strtok(0,brkset);
    }
}
```

```
This\0is 1just2a3test#.
This\0is\01just2a3test#.
This\0is\01just\0a3test#.
This\0is\01just\0a\0test#.
This\0is\01just\0a\0test\0#.
```

```
This
is
just
a
test
```

The Dependencies of **strtok**



- What “wrong” here?

What's "Wrong" with **strtok**?

- It *modifies* the original search string
  - a *side effect!*
- It keeps *static data*
  - *another* side effect!
  - *shared* among *all calls* to **strtok**
  - calls from independent clients can't be interleaved
- Different calls to **strtok** have *different semantics*
  - when you pass 0 (NULL), it picks up where it left off
  - it is essentially *2 different functions*

An Improved Tokenizer

- Will *not* modify the caller's data
  - but how can we keep track of where we are?
- Will *not* use shared (static) data to track its state
  - but where will it put it?
- Will separate *initialization* from *iteration*
  - *different calls* for different actions
  - so we will need more that one function!

Another Try at **strtok**

```
struct Tokenizer {
    const char* search;
    const char* brkset;
    int pos;
};

Tokenizer* init_tok(const char* s, const char* brkset);

string next_token(Tokenizer* tok);

void del_tok(Tokenizer* tok);
```

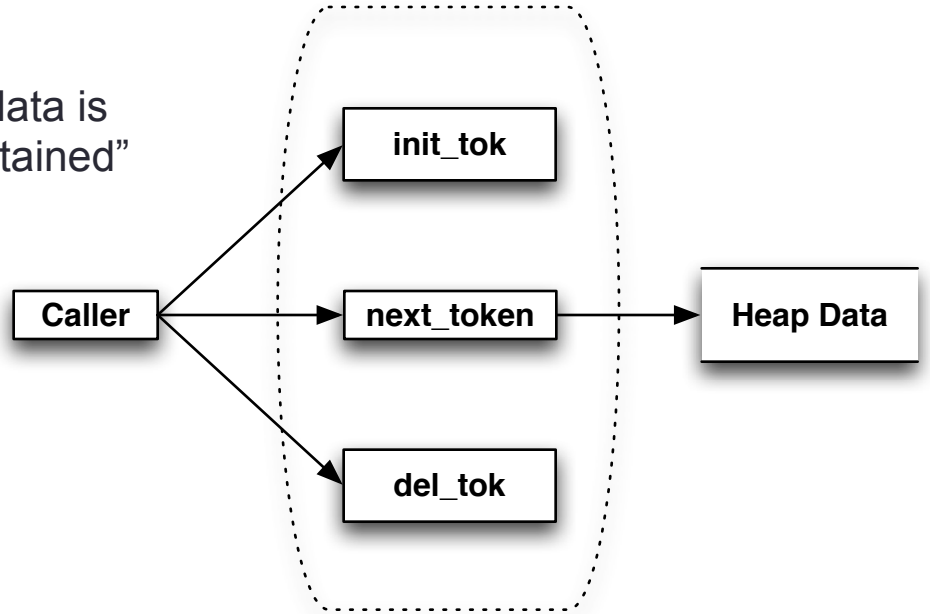
Code is available in *strtok2.cpp* if you want to see it.

Using Tokenizer

```
int main() {
    char search[BUFSIZ];
    strcpy(search,"This is 1just2a3test#.");
    char brkset[] = " \t\n\r\f\v`~!@#$%^&*()-_”
                    "=+;:'\",<.>/?01234567890";
    Tokenizer* tok = init_tok(search,brkset);
    string word = next_token(tok);
    while (!word.empty()) {
        cout << word << endl;
        word = next_token(tok);
    }
    del_tok(tok);
}
```

The Dependencies of Tokenizer

But the data is
“self-contained”



Using a *class* would be better
of course (see *strtok3.cpp*)

Reentrant Functions

- The problem with threads: race conditions on *shared data*
  - Using *critical sections* for shared data is a topic for another day
- **strtok** is *not thread-safe*:
  - it uses *static data* which is *shared by its very nature*
  - but each thread needs its *own copy*; FAIL
- Thread-safe functions must be *reentrant*
  - they “start from scratch” on each call;
  - no external data; no state saved between calls

Generators

- Generators are special functions that:
  - *save state* between calls (in each function activation)
  - pick up *where they left off* when resumed
  - like *iterators* do
- No need for us to manage the state at all
- Python supports generators:
  - and they can be easily simulated with *iterators* in other languages

A Tokenizer Generator

```
def tokenizer(s,brkset):
    pos = 0
    while pos < len(s):
        # Skip break characters
        start = pos
        while start < len(s) and s[start] in brkset:
            start += 1
        if start == len(s): raise StopIteration

        # Find next break character
        stop = start
        while stop < len(s) and s[stop] not in brkset:
            stop += 1
        pos = stop
        yield s[start:stop]   # ←

search = "This is 1just2a3test#."
brkset = " \t\n\r\f\v`~!@#$%^&*()-_=+;:'\",<.>/?01234567890"
for word in tokenizer(search,brkset):
    print word
```

A Numeric Example

- Classic algorithm for **sqrt(x)**:
  - start with an initial guess, g_1
  - compute next guess as:
    - $g_2 = \frac{1}{2} (g_1 + x/g_1)$
  - continue until the difference between guesses is “small”

```
def mysqrt(x,g1,tol):
    g2 = (g1 + x/g1)/2.0
    while abs(g2 - g1) > tol:
        g1 = g2
        g2 = (g1 + x/g1)/2.0
    return g2
```

Is There Room for Improvement?

- Depends…

- There is no coupling to non-local data ✔
- There is no shared data ✔

- Hmmm. Maybe it's "perfect"

There are 2 Things At Play in **mysqrt**

- 1) Generating the next guess
- 2) Checking a condition governing the iteration
- Can these be *separated*?
- And why would we want to separate them?

Loosening the Coupling

- Iterating until a stopping condition is obtained is a very common operation
- Let's feed the sequence of guesses to a generic iteration procedure
  - generators make that easy
- Thus we will *loosen* the coupling between the two actions
  - by making the sequence a *parameter* to the iteration

The Sequence Generator

- An *unbounded* sequence

```
def sqrt_seq(x,g):
    yield g
    while True:
        g = (g + x/g) / 2.0
        yield g
```

The Iteration Procedure

- It decides when to quit

```
def iterate(seq, tol):
    last = seq.next()
    current = seq.next()
    while (abs(current-last) > tol):
        last = current
        current = seq.next()
    return current
```

Using the New Arrangement

```
def mysqrt(x,g1,tol):
    return iterate(sqrt_seq(x,g1),tol)

print mysqrt(2.0,1.0,.001)
```

- We can now *reuse* **iterate** on any sequence!
- (*sqrt.cpp* has a C++ version)

Coupling Guidelines

- *Eliminate* unnecessary dependencies
- *Minimize* and *localize* the number of necessary dependencies
- Strive to only communicate through *interfaces*
- Minimize (or eliminate) *shared data*
- Minimize the number of *parameters* in an interface

Maxim

- “As little coupling as possible, but no less.”

Cohesion

- A measure of *integrity*
  - “the task, the whole task, and nothing but the task”
- Functions that are cohesive perform a *single*, well-defined (logical) *task*
- There is an *inverse relationship* between cohesion and coupling
  - if we *think cohesion*, coupling will take care of itself!

Coroutines

- A *coroutine*, like a generator, can be *paused* and *resumed*
  - *multiple* entry and exit points
  - ideal for *cooperating procedures* and *simulations*
- Coroutines can also *receive input* when they resume
  - Coroutines support *cooperative multitasking* (aka “Green Threads”)
- Functions and procedures are *special cases* of coroutines
  - they have only one entry point
  - data is only received at the time of call

Coroutine Example

Write a *coroutine* named **changes** that receives numbers one at a time from clients (via **send**) and returns a list of the (zero-based) *positions* where changes in sign are detected as they are encountered. Positive numbers, negative numbers, and zero are considered of "different signs". The number of inputs is *unbounded*. An example:

```
f = changes()
f.next()        # Returns [] (ignored)
nums = [1,2,0,1,-1,-2,3]
for n in nums:
    print f.send(n)
```

Output:

```
[]
[]
[2]
[2, 3]
[2, 3, 4]
[2, 3, 4]
[2, 3, 4, 6]
```

The Coroutine **changes**

```
def changes():
    def sign(n):
        return -1 if n < 0 else 1 if n > 0 else 0

    result = []
    last = (yield result)          # output/input
    index = 0
    while True:
        curr = (yield result)      # output/input
        index += 1
        if sign(last) != sign(curr):
            result.append(index)
        last = curr
```

Optional Interlude

- If time and interest allow:
  - look at a /*…*/ *comment extractor* coroutine
  - file *c_comment.py*

Nested Functions

- Python and D support *nested function* definitions
  - **sign** is *nested* inside **changes** on 2 slides back
- What is a possible advantage for this?

Minimal Scoping

- Identifiers should have the *smallest scope* possible
- *Reduced visibility* promotes *reduced coupling*
- Will see more in the next section…

Functions and Reuse

- We've seen that function *reuse* can result from a *separation of concerns*
  - when we separated **iterate** from **mysqrt**
- Reusable Code is:
  - *generic* (specialized by parameters)
  - *encapsulated* (few external dependencies)
  - *flexible* (can be customized by more than just data parameters)
- Minimal coupling/Maximal Cohesion ➟ Reusability

FUNCTIONS AS FIRST-CLASS OBJECTS

Parameterizing the Break Set

- Let's enhance our **tokenizer**
  - allow *users* to specify the set of *break characters*
  - by passing a *function* to the tokenizer
- This allows *maximum flexibility* in specifying what unacceptable characters are
  - *computations* are more *generic* than sets of data
  - e.g., they can use **if-else** logic for more complex conditions

Using a Break-Set Function

```
def tokenizer(s,brkf):
    pos = 0
    while pos < len(s):
        # Skip break characters
        start = pos
        while start < len(s) and brkf(s[start]):
            start += 1
        if start == len(s): raise StopIteration

        # Find next break character
        stop = start
        while stop < len(s) and not brkf(s[stop]):
            stop += 1
        pos = stop
        yield s[start:stop]
```

Using the New Tokenizer

```
search = "This is 1just2a3test#."

def badchars(c):
    return not c.isalpha()

for word in tokenizer(search,badchars):
    print word
```

This
is
just
a
test

Being More Direct

- Why not specify the *desired characters* directly?

```
def tokenizer(s,accept):
    pos = 0
    while pos < len(s):
        # Skip break characters
        start = pos
        while start < len(s) and not accept(s[start]):
            start += 1
        if start == len(s): raise StopIteration

        # Find next break character
        stop = start
        while stop < len(s) and accept(s[stop]):
            stop += 1
        pos = stop
        yield s[start:stop]
```

Being More Direct (cont.)

```
search = "This isn't 1just2a3test#."

def goodchars(c):
    return c.isalpha() or c == "'"

for word in tokenizer(search,goodchars):
    print word
```

This
isn't
just
a
test

Higher-Order Functions

- A *Higher-Order Function*:
  - accepts functions as *parameters*, and/or
  - returns a function as a *result*

- **tokenizer** is a higher-order function
  - C++, C#, and D support this with *function objects*
  - C# and D also support this with *delegates*

- Functions in some languages are *first-class objects*
  - they can be copied and created *on-the-fly*

sort is a Higher-Order Function

- You can customize its *comparison* functionality
- C++:
  - **sort(start_pos, endp1_pos, *comparator*$_{opt}$)**
- Python
  - **sort(sequence, *comparator*$_{opt}$)**
- Java:
  - **Arrays.sort(array, *comparator*$_{opt}$)**
  - **Collections.sort(list, *comparator*$_{opt}$)**

Customizing **sort** in Python

Descending Order

```
def desc(a,b):
    return 1 if a < b else -1 if a > b else 0

stuff = [1,2,3,4,5]
stuff.sort(desc)
print stuff     # [5, 4, 3, 2, 1]
```

Customizing **sort** in C++

```
bool desc(int a, int b) {
    return b < a;
}

int main() {
    vector<int> stuff = {1,2,3,4,5};
    sort(stuff.begin(), stuff.end(), &desc);
    for (int i: stuff)
        cout << i << ' ';
    cout << endl;
}
```

Anonymous Functions

- The name **desc** is not important
- We can create the function *on-the-fly*
  - the epitome of *minimal scoping*
- Python, D, C++ and C# support:
  - *lambda expressions*
  - *function objects* (aka “functors”)

Using Lambda Expressions

Python:

```
stuff = [1,2,3,4,5]
stuff.sort(lambda a,b: 1 if a < b else -1 if a > b else 0)
print stuff     # [5, 4, 3, 2, 1]
```

C++ 2011:

```
int main() {
    vector<int> stuff = {1,2,3,4,5};
    sort(stuff.begin(), stuff.end(), [](int a,int b){return a > b;});
    for (int i: stuff)
        cout << i << ' ';
    cout << endl;
}
```

Using a C++ Function Object

```
int main() {
    vector<int> stuff = {1,2,3,4,5};
    sort(stuff.begin(), stuff.end(), greater<int>());
    for (int i: stuff)
        cout << i << ' ';
    cout << endl;
}
```

Returning Functions as Values

- Functions can be *created* and *returned* from inside a host function
- Useful when parameters to the host function *customize* the *result function*
- The result functions can use the host function's environment
  - via a *closure*
  - a *packaging* of code with its referencing environment

Python Closure Example

- An “add n” function

```
def addn(n):
    return lambda x: x+n

add3 = addn(3)
print add3(1)        # 4
print add3(2)        # 5
```

C++ 2011 Version

```
#include <functional>
#include <iostream>
using namespace std;

function<int(int)> addn(int n) {
    return [=](int x){return x + n;};
}

int main() {
    auto add3 = addn(3);
    cout << add3(1) << endl;      // 4
    cout << add3(2) << endl;      // 5
}
```

D Version

```
import std.stdio;

auto addn(int n) {
    return (int x){return x + n;};  // a delegate
}

void main() {
    auto add3 = addn(3);
    writeln(add3(1));       // 4
    writeln(add3(2));       // 5
}
```

Delegates in D

- As in C#, a delegate in D can be coupled with:
  - an object (for non-static methods)
  - a class (for static methods)

- In D, delegates can *also* be coupled with a *function closure*
  - which is moved from the stack to the *garbage-collected heap*
  - like we just saw with **addn**

List Processing Functions

- A staple of *functional programming*
- *Higher-level* coding than using explicit loops
  - avoid creating your own function
- **map** (aka **transform**)
  - *applies* a function to each list element (returns a new list)
- **filter** (aka **copy_if**)
  - *collects* list elements satisfying a predicate function
- **reduce** (aka **accumulate**, **foldl**)
  - i.e., “reduces” the list to a computed result (sum, max, etc.)

Python List Processing Examples

```
urls = ["http://python.org","http://stickyminds.com",
        "http://sqe.com","http://signsoflife.me”,
        "http://uvu.edu"]

short_urls = map(lambda s: s[len("http://"):],urls)
print short_urls
dotcom = filter(lambda s: s.endswith(".com"),urls)
print dotcom
anyedu = reduce(lambda sofar,s: sofar or s.endswith(".edu"),
                urls,False)
print anyedu
```

''' Output:
['python.org', 'stickyminds.com', 'sqe.com', 'signsoflife.me', 'uvu.edu']
['http://stickyminds.com', 'http://sqe.com']
True
'''

short_urls in C++

```
 vector<string> urls = {
    "http://python.org","http://stickyminds.com",
    "http://sqe.com", "http://signsoflife.me",
    "http://uvu.edu"};

// Remove http://
vector<string> short_urls;
size_t prefix_len = strlen("http://");
transform(urls.begin(),urls.end(),back_inserter(short_urls),
    [=](string s){return s.substr(prefix_len);});
for (string s: short_urls)
    cout << s << endl;
```

Partial Function Application

- You can send only a *subset* of arguments to a function
  - leaving the *rest* for *later*
  - a new, *temporary function* waiting for the *other args* is returned
  - allows *customization* of functions for later *reuse*

- aka “Currying”

Currying in Python

```
import functools

# Store the accumulator function
add = functools.partial(reduce,lambda x,y:x+y)
print add([1,2,3])           # 6 (0 initializer assumed)
print add([4,5,6],100)       # 115

# Fix the list
add123 = functools.partial(add,[1,2,3])
print add123(0)              # 6
print add123(100)            # 106
```

A C++2011 Version

```
int main() {
    vector<int> nums = {1,2,3}, nums2 = {4,5,6};
    typedef vector<int>::iterator Iter;
    auto add = bind(&accumulate<Iter,int,function<int(int,int)>>,
                    _1,_2,_3,plus<int>());
    cout << add(nums.begin(),nums.end(),0) << endl;     // 6
    cout << add(nums2.begin(),nums2.end(),100) << endl; // 115

    auto add123 = bind(add,nums.begin(),nums.end(),_1);
    cout << add123(0) << endl;                          // 6
}
```

A D Example

```
void main() {
    int[] nums = [1,2,3], nums2 = [4,5,6];
    alias reduce!("a+b") add;
    writeln(add(nums));          // 6

    alias curry!(add,100) addto100;
    writeln(addto100(nums2));    // 115
}
```

Binding Args via Another Function

In D

```
int f(int a, int b, int c) {
    return a + b*c;
}

void main() {
    // Fix b = 5
    auto g = (int a, int c) {return f(a,5,c);};
    writeln(g(4,6));    // 4 + 5*6 = 34
}
```

Pure Functions

- Nice feature: *Referential Transparency*
  - whenever you call them with the *same arguments*, you always get the *same answer*
  - assists in writing *correct programs*
  - no surprises via coupling to enclosing scopes
- All functions in functional languages are pure:
  - Haskell, FP subset of ML
  - Supported in D

A Pure Function Example

- The proverbial Fibonacci Number function:
  - D *ensures* that the function is *self-contained*

```
pure int fib(int n) {
    if (n == 0 || n == 1)
        return n;
    int a = 1, b = 1;
    foreach (i; 2..n) {
        auto t = b;
        b += a;
        a = t;
    }
    return b;
}
```

POLYMORPHIC FUNCTIONS

Polymorphism

- A function is *polymorphic* if at least one of its parameters can receive arguments of *different types*

- Ad Hoc polymorphism ("bounded" polymorphism)
  - the permissible set of types is *fixed* with the function definition(s)
  - *implicit conversions* (aka *coercions*)
  - *overloading*

- Universal polymorphism ("unbounded")
  - the permissible set of types is *not fixed*
  - *generics/templates* ("parametric polymorphism")
  - *method overrides* ("subtype polymorphism")

Duck Typing

- A type of *parametric polymorphism* found in C++ Templates, C# 4.0 and dynamically typed languages
  - Ruby, Python, Perl, PHP, etc.
- aka “Implicit Interfaces”
  - as long as the supplied type has the *expected operations*, all is well
  - no need to require implementation of an explicit interface
    - like Java generics do
- Parametric Polymorphism comes “for free” in dynamic languages

Parametric Polymorphism in Python

```
def h(x):
    return x + x

# g calls f on x:
def g(f, x):          # f must be a function
    return f(x)

print g(h,3)          # 6
print g(h,'two')      # twotwo
#print g(2,3)         # error: 2 is not callable
```

A C++ Version

```
template<typename T>
T h(T t) {
    return t + t;
}

template<typename F, typename T>
T g(F f, T t) {
    return f(t);
}

int main() {
    cout << g(&h<int>,3) << endl;                    // 6
    cout << g(&h<string>,string("two")) << endl;     // twotwo
}
```

Contract Programming

A Perspective for Polymorphic Interfaces

- Methods are *contracts* with users
- Users must meet *pre-conditions* of a method
  - what the method *requires* of the client
  - parameter in a certain range, for example
- Method *guarantees* certain *post-conditions*
  - but *only* if the pre-conditions were met

Parties in Contracts

Clients and Suppliers

- **Clients** *must* satisfy pre-conditions
  - Think of preconditions as the *price a customer pays* for a service
- **Suppliers** *must* satisfy post-conditions
  - Think of postconditions as the *service provided to a customer*
- This affects *inheritance*…

Liskov Substitution Principle

- A technique for designing *function overrides* in a *hierarchy*
  - ensures correct subtype polymorphism

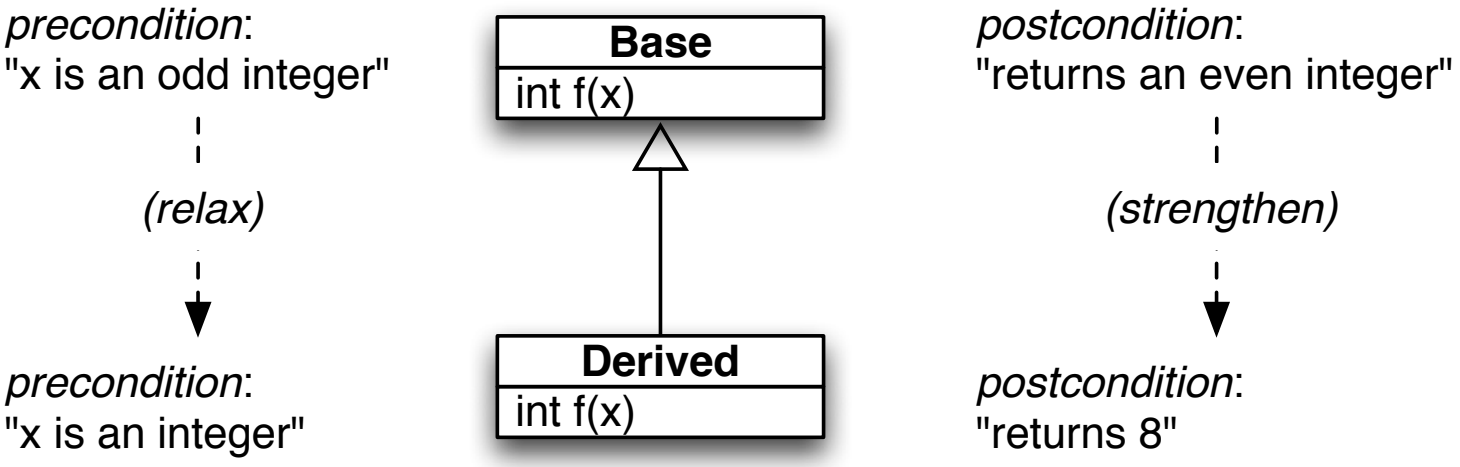
- Liskov Substitution Principle:
  - "Let q(x) be a property provable about objects x of type T. Then q(y) should be true for objects y of type S where S is a subtype of T."
  - *Derived objects should be substitutable for base objects*

- This principle must be followed when overriding functions
  - Derived classes must *not* change the "rules of the game"

Contracts and Inheritance

- Contracts are set by the *base class interface*
  - Derived classes must *obey* the base contract
  - Otherwise *substitutability* is *compromised*
- Pursuing the contractor-customer analogy:
  - Subcontractors must *not charge more* than originally agreed upon
  - Subcontractors must *deliver at least* what was agreed upon
- Clients program to the *contract*
  - By using and understanding the *base class interface* and its *conditions*
  - And by using *base/interface pointers*
    - and letting *polymorphism* do its work invisibly

Sample Contract Specification



D Contract Implementation

Base Class

```
class Base {
public:
    int f(int x)
    in {
        assert(x % 2 == 1);
    }
    out (retval) {
        assert(retval % 2 == 0);
    }
    body {
        int retval = x*2;
        return retval;
    }
}
```

D Contract Implementation

Derived Class

```
class Derived : Base {
public:
    int f(int x)
    in {
        assert(is(typeof(x) : int));
    }
    out (retval) {
        assert(retval == 8);
    }
    body {
        int retval = 8;
        return retval;
    }
}
```

D Contract Semantics with Inheritance

- All preconditions are **OR**'ed together
- All postconditions are **AND**'ed
- Using *short-circuit evaluation*
  - from the top of the hierarchy down

Contract Programming

Summary

- "The problem for instances of B is how to be perfectly substitutable for instances of A. The only way to guarantee type safety and substitutability is to be *equally or more liberal than A on inputs*, and to be *equally or more strict than A on outputs*." – *Wikipedia*

- *"Require no more; Promise no less"*

RESOURCE-FRIENDLY FUNCTIONS

Resource Management

- Most resources are used with an *acquire-release* pattern
  - e.g., locks, connections, etc.
- Exceptions *interrupt* this flow
  - but resources still need to be released!
- Solution: *automatic cleanup* during stack unwinding
  - **finally**
  - *RAII* (put release code in C++ *destructors* or C# **IDisposable** objects with **using**)
  - **with** clauses in Python
  - **scope** statements in D

An Unsafe Function

```
void g() {
   risky_op1();      // May acquire resources…
   risky_op2();      // "
   risky_op3();      // "
   writeln("g succeeded");
}
```

Assume further that these functions must run to completion or not at all (i.e., *transactionally*).

An Unsavory Solution

```
void g() {
    risky_op1();
    try {
        risky_op2();
    }
    catch (Exception x) {
        undo_risky_op1();      // Back-out op1
        throw x;               // Rethrow exception
    }
    try {
        risky_op3();
        writeln("g succeeded");
    }
    catch (Exception x) {
        // Back-out op1 and op2 in reverse order
        undo_risky_op2();
        undo_risky_op1();
        throw x;
    }
}
```

D's **scope** Statement

```
void g() {
    risky_op1();
    scope(failure) undo_risky_op1();
    risky_op2();
    scope(failure) undo_risky_op2();
    risky_op3();
    writeln("g succeeded");
}
```

A LIFO Transaction!

PARAMETER PASSING TECHNIQUES

Many Ways To Pass a Parameter

- More than we need!

- 1) by value (“copy in”; **in**)
- 2) by result (“copy out”; **out**)
- 3) by value-result (“copy in–copy out”; **inout** or **in out**)
- 4) by reference (**ref**, **&** in C++)
- 5) by object reference (Java, C#, D)
- 6) by name (weird relic from lambda calculus)
- 7) by need (by *name* “done right”)

- 1-3 make *copies*
- 6 and 7 are *lazy*

Copies of Arguments

- Fine for *small objects*
- Often more *efficient* to pass *by reference*
  - but by-reference allows making *non-local changes*
  - *best of both worlds*: C++'s pass by **const** reference
- Pass *by-result issue*: also makes non-local changes
  - nearly identical semantics as pass by-reference
  - often used for returning *multiple values*
  - consider using *tuples* instead

OUT Parameters in D

```
void plus(int a, int b, out int c)
{
    writeln(c);       // 0
    c = a+b;
}

void main() {
    int x = 3;
    int y = 4;
    int z;
    plus(x, y, z);
    writeln(z);       // 7
}
```

Passing By Reference

- Two Types…

- Passing an *Object Reference* (aka pass by *sharing*)
  - as in Java, C#, and D
  - or a pointer in C++
  - changes to *object attributes* persist
  - but object-id does not change

- True pass by-reference
  - as in C++
  - an *lvalue* is passed; the parameter is an *alias* for the original arg
  - changes persist in calling context

Reference Parameters

Transparent access to argument

```
void f(int x) {cout << &x << endl;}

void g(int& x) {cout << &x << endl;}

int main() {
    int n;
    cout << &n << endl;
    f(n);
    g(n);
}

/* Output:
0x7fff5fbff9ac
0x7fff5fbff98c
0x7fff5fbff9ac
*/
```

Pass by Reference

```
void swap(int& x, int& y)
{
    int temp = x;
    x = y;
    y = temp;
}

int main()
{
    int i = 1, j = 2;

    swap(i,j);
    cout << "i == " << i << ", j == " << j;
}

/* Output:
i == 2, j == 1
*/
```

Aliasing

```
void sigsum(int& n, int& ans) {
  ans = 0;
  int i = 1;
  while (i <= n)
      ans += i++;
}

int f() {
  int x,y;
  x = 10;
  sigsum(x,y);
  return y;
}

int g() {
  int x;
  x = 10;
  sigsum(x,x);
  return x;
}
```

f() returns 55
g() returns 0

Value-Result ≠ Reference

An Aliasing Issue

```
void f(ref int x, ref int y) {
    x = 1;
    y = 3;
}

void main() {
    int[2] a = [0,2];
    f(a[0],a[a[0]]);
    writeln(a);
}
```

Lazy Evaluation

- A hallmark of *functional programming*
- Allows for a high-degree of *separation* between caller and callee
- Results in simple, efficient code
- Parameters are *not evaluated* at the call site
  - but only *when and if* they're *used* in the called function
  - a little function ("thunk") is actually passed

Lazy Evaluation in D

```
void f(bool flag, lazy void exp) {
    if (flag)
        exp();
}

void main() {
    int x;
    f(false,x=3);
    writeln(x);     // 0
    f(true,x=3);
    writeln(x);     // 3
}
```

Implementing a Short-Circuit AND

```
bool myand(lazy int x, lazy int y) {
    if (!x)
        return false;
    if (!y)
        return false;
    return true;
}

void main() {
    int x = 0;
    writeln(myand(0,1/x));  // false (1/x not evaluated)
}
```

Pass By-Need

- A *caching thunk* is passed
- It is only evaluated *once*, and the result is *cached* for subsequent accesses
  - no non-local changes are possible
- Supported by Haskell
  - and to some degree by Scheme
  - can be simulated in languages with a *function-call* operator

Python Example

- We create a function *decorator* (“wrapper”) that caches the parameters when they are first evaluated

```
def memo(f):
    "A memoizing decorator"
    f.cache = {}
    def wrapper(*args):
        if args in f.cache:
            print "found in cache"  # trace to test caching
            return f.cache[args]
        else:
            f.cache[args] = value = f(*args)
            return value
    return wrapper
```

Using **memo**

```
from memo import *

def exc(g):
    print g()
    print g()

exc(memo(lambda : "hello"))

""" Output:
hello
found in cache
hello
"""
```

THE END!