Control Flow

COMS W4115

Prof. Stephen A. Edwards
Spring 2007
Columbia University
Department of Computer Science

Control Flow

“Time is Nature’s way of preventing everything from happening at once.”

There are at least seven manifestations:

1. Sequencing	foo(); bar();
2. Selection	if (a) foo();
3. Iteration	while (i<10) foo(i);
4. Procedures	foo(10,20);
5. Recursion	foo(int i) **{** foo(i-1); **}**
6. Concurrency	foo() || bar()
7. Nondeterminism	do a **->** foo(); [] b **->** bar();

Ordering Within Expressions

What code does a compiler generate for

```
a = b + c + d;
```

Most likely something like

```
tmp = b + c;
a = tmp + d;
```

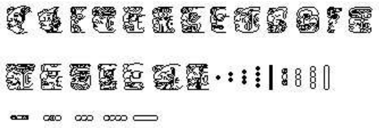
(Assumes left-to-right evaluation of expressions.)

Order of Evaluation

Why would you care?

Expression evaluation can have side-effects.

Floating-point numbers don’t behave like numbers.



Mayan numbers

Side-effects

```
int x = 0;

int foo() { x += 5; return x; }

int a = foo() + x + foo();
```

What’s the final value of a?

Side-effects

```
int x = 0;

int foo() { x += 5; return x; }

int a = foo() + x + foo();
```

GCC sets a=25.

Sun’s C compiler gave a=20.

C says expression evaluation order is implementation-dependent.

Side-effects

Java prescribes left-to-right evaluation.

```
class Foo {
  static int x;
  static int foo() { x += 5; return x; }
  public static void main(String args[]) {
    int a = foo() + x + foo();
    System.out.println(a);
  }
}
```

Always prints 20.

Number Behavior

Basic number axioms:

$a + x$	$=$	a if and only if $x = 0$	Additive identity
$(a + b) + c$	$=$	$a + (b + c)$	Associative
$a(b + c)$	$=$	$ab + ac$	Distributive

Misbehaving Floating-Point Numbers

1e20 + 1e-20 = 1e20

1e-20 $\ll$ 1e20

(1 + 9e-7) + 9e-7 $\neq$ 1 + (9e-7 + 9e-7)

9e-7 $\ll$ 1, so it is discarded, however, 1.8e-6 is large enough

$1.00001(1.000001 - 1) \neq 1.00001 \cdot 1.000001 - 1.00001 \cdot 1$

$1.00001 \cdot 1.000001 = 1.00001100001$ requires too much intermediate precision.

What’s Going On?

Floating-point numbers are represented using an exponent/significand format:

$$1 \quad \underbrace{10000001}_{\text{8-bit exponent}} \quad \underbrace{01100000000000000000000}_{\text{23-bit significand}}$$

$$= -1.011_2 \times 2^{129-127} = -1.375 \times 4 = -5.5.$$

What to remember:

$$\underbrace{1363.4568}_{\text{represented}}\underbrace{46353963456293}_{\text{rounded}}$$

What’s Going On?

Results are often rounded:

$$\begin{array}{r} 1.00001000000 \\ \times 1.00000100000 \\ \hline 1.000011\,\underbrace{00001}_{\text{rounded}} \end{array}$$

When $b \approx -c$, $b + c$ is small, so $ab + ac \neq a(b + c)$ because precision is lost when ab is calculated.

Moral: Be aware of floating-point number properties when writing complex expressions.

Short-Circuit Evaluation

When you write

```
if (disaster_could_happen)
  avoid_it();
else
  cause_a_disaster();
```

`cause_a_disaster()`is not called when `disaster_could_happen`is true.

The *if* statement evaluates its bodies lazily: only when necessary.

Short-Circuit Evaluation

The section operator **? :** does this, too.

```
cost =
 disaster_possible ? avoid_it() : cause_it();
```

`cause_it` is not called if `disaster_possible`is true.

Logical Operators

In Java and C, Boolean logical operators “short-circuit” to provide this facility:

```
if (disaster_possible || case_it()) { ... }
```

`cause_it()` only called if `disaster_possible`is false.

The **&&** operator does the same thing.

Useful when a later test could cause an error:

```
int a[10];

if (i => 0 && i < 10 && a[i] == 0) { ... }
```

Short-Circuit Operators

Not all languages provide short-circuit operators. Pascal does not.

C and Java have two sets:

Logical operators **||** **&&** short-circuit.

Boolean (bitwise) operators **|** **&** do not.

Unstructured Control-Flow

Assembly languages usually provide three types of instructions:

Pass control to next instruction:

```
add, sub, mov, cmp
```

Pass control to another instruction:

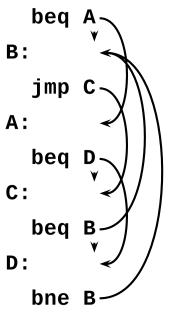
```
jmp rts
```

Conditionally pass control next or elsewhere:

```
beq bne blt
```

Unstructured Control-Flow

So-called because it’s easy to create spaghetti:



```
  beq A
B:
  jmp C
A:
  beq D
C:
  beq B
D:
  bne B
```

Structured Control-Flow

The “object-oriented languages” of the 1960s and 70s.

Structured programming replaces the evil goto with structured (nested) constructs such as

if-then-else
for
while
do .. while
break
continue
return

Gotos vs. Structured Programming

A typical use of a goto is building a loop. In BASIC:

```
10 print I
20 I = I + 1
30 IF I < 10 GOTO 10
```

A cleaner version in C using structured control flow:

```
do {
  printf("%d\n", i);
  i = i + 1;
} while ( i < 10 )
```

An even better version

```
for (i = 0 ; i < 10 ; i++) printf("%d\n", i);
```

Gotos vs. Structured Programming

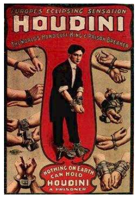
Break and continue leave loops prematurely:

```
for ( i = 0 ; i < 10 ; i++ ) {
  if ( i == 5 ) continue;
  if ( i == 8 ) break;
  printf("%d\n", i);
}
Again: if (!(i < 10)) goto Break;
  if ( i == 5 ) goto Continue;
  if ( i == 8 ) goto Break;
  printf("%d\n", i);
Continue: i++; goto Again;
Break:
```

Escaping from Loops

Java allows you to escape from labeled loops:

```
a: for (int i = 0 ; i < 10 ; i++)
     for ( int j = 0 ; j < 10 ; j++) {
        System.out.println(i + "," + j);
        if (i == 2 && j == 8) continue a;
        if (i == 8 && j == 4) break a;
     }
```



Gotos vs. Structured Programming

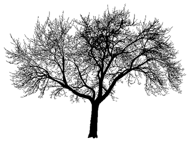
Pascal has no “return” statement for escaping from functions/procedures early, so goto was necessary:

```
procedure consume_line(var line : string);
begin
  if line[i] = '%' then goto 100;
  (* .... *)
100:
end
```

In C and many others, return does this for you:

```
void consume_line(char *line) {
  if (line[0] == '%') return;
}
```

Multi-way Branching



```
switch (s) {
case 1: one(); break;
case 2: two(); break;
case 3: three(); break;
case 4: four(); break;
}
```

Switch sends control to one of the case labels. Break terminates the statement.

Implementing multi-way branches

```
switch (s) {
case 1: one(); break;
case 2: two(); break;
case 3: three(); break;
case 4: four(); break;
}
```

Obvious way:

```
if (s == 1) { one(); }
else if (s == 2) { two(); }
else if (s == 3) { three(); }
else if (s == 4) { four(); }
```

Reasonable, but we can sometimes do better.

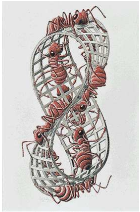
Implementing multi-way branches

If the cases are *dense*, a branch table is more efficient:

```
switch (s) {
case 1: one(); break;
case 2: two(); break;
case 3: three(); break;
case 4: four(); break;
}
```

```
labels l[] = { L1, L2, L3, L4 }; /* Array of labels */
if (s>=1 && s<=4) goto l[s-1];   /* not legal C */
L1: one(); goto Break;
L2: two(); goto Break;
L3: three(); goto Break;
L4: four(); goto Break;
Break:
```

Recursion and Iteration



Consider computing

$$\sum_{i=0}^{10} f(i)$$

In C, the most obvious evaluation is iterative:

```
double total = 0;
for ( i = 0 ; i <= 10 ; i++ )
  total += f(i);
```

Recursion and Iteration

$$\sum_{i=0}^{10} f(i)$$

But this can also be defined recursively

```
double sum(int i)
{
  double fi = f(i);
  if (i <= 10) return fi + sum(i+1);
  else return fi;
}

sum(0);
```

Recursion and Iteration

Grammars make a similar choice:

Iteration:

```
clist : item ( "," item )* ;
```

Recursion:

```
clist : item tail ;

tail : "," item tail
     | /* nothing */
     ;
```

Tail-Recursion and Iteration

```
int gcd(int a, int b) {
  if ( a==b ) return a;
  else if ( a > b ) return gcd(a-b,b);
  else return gcd(a,b-a);
}
```

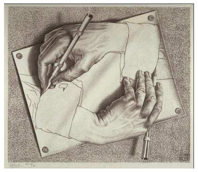
Notice: no computation follows any recursive calls.

Stack is not necessary: all variables “dead” after the call.

Local variable space can be reused. Trivial since the collection of variables is the same.

Tail-Recursion and Iteration

```
int gcd(int a, int b) {
  if ( a==b ) return a;
  else if ( a > b ) return gcd(a-b,b);
  else return gcd(a,b-a);
}
```



Can be rewritten into:

```
int gcd(int a, int b) {
start:
  if ( a==b ) return a;
  else if ( a > b ) a = a-b; goto start;
  else b = b-a; goto start;
}
```

Tail-Recursion and Iteration

Good compilers, especially those for functional languages, identify and optimize tail recursive functions.

Less common for imperative languages.

But gcc -O was able to rewrite the gcd example.

Applicative- and Normal-Order Evaluation

```
int p(int i) { printf("%d ", i); return i; }

void q(int a, int b, int c)
{
  int total = a;
  printf("%d ", b);
  total += c;
}
```

What is printed by

```
q( p(1), 2, p(3) );
```

Applicative- and Normal-Order Evaluation

```
int p(int i) { printf("%d ", i); return i; }
void q(int a, int b, int c)
{
  int total = a;
  printf("%d ", b);
  total += c;
}
q( p(1), 2, p(3) );
```

Applicative: arguments evaluated before function is called.

Result: 1 3 2

Normal: arguments evaluated when used.

Result: 1 2 3

Applicative- vs. and Normal-Order

Most languages use applicative order.

Macro-like languages often use normal order.

```
#define p(x) (printf("%d ",x), x)
#define q(a,b,c) total = (a), \
   printf("%d ", (b)), \
   total += (c)

q( p(1), 2, p(3) );
```

Prints 1 2 3.

Some functional languages also use normal order evaluation to avoid doing work. “Lazy Evaluation”

Argument Order Evaluation

C does not define argument evaluation order:

```
int p(int i) { printf("%d ", i); return i; }
int q(int a, int b, int c) {}

q( p(1), p(2), p(3) );
```

Might print 1 2 3, 3 2 1, or something else.

This is an example of *nondeterminism*.

Nondeterminism

Nondeterminism is not the same as random:

Compiler usually chooses an order when generating code.

Optimization, exact expressions, or run-time values may affect behavior.

Bottom line: don’t know what code will do, but often know set of possibilities.

```
int p(int i) { printf("%d ", i); return i; }
int q(int a, int b, int c) {}
q( p(1), p(2), p(3) );
```

Will *not* print 5 6 7. It will print one of

1 2 3, 1 3 2, 2 1 3, 2 3 1, 3 1 2, 3 2 1

Nondeterminism

Nondeterminism lurks in most languages in one form or another.

Especially prevelant in concurrent languages.

Sometimes it's convenient, though:

```
if a >= b -> max := a
[] b >= a -> max := b
fi
```

Nondeterministic (irrelevant) choice when a=b.

Often want to avoid it, however.

Implementing Inheritance

Simple: Add new fields to end of the object

Fields in base class always at same offset in derived class

Consequence: Derived classes can never remove fields

C++

```
class Shape {
  double x, y;
};

class Box : Shape {
  double h, w;
};
```

Equivalent C

```
struct Shape {
  double x, y;
};

struct Box {
  double x, y;
  double h, w;
};
```

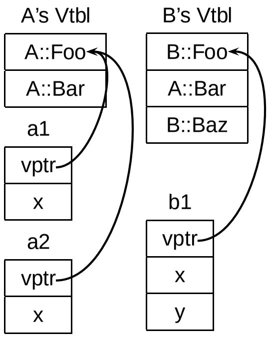
Virtual Functions

```
class Shape {
  virtual void draw(); // Invoked by object's class
};                     // not its compile-time type.
class Line : public Shape {
  void draw();
};
class Arc : public Shape {
  void draw();
};

Shape *s[10];
s[0] = new Line;
s[1] = new Arc;
s[0]->draw();    // Invoke Line::draw()
s[1]->draw();    // Invoke Arc::draw()
```

Virtual Functions

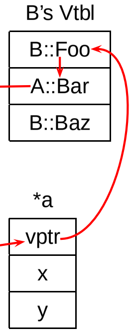
The Trick: Add a "virtual table" pointer to each object.

```
struct A {
  int x;
  virtual void Foo();
  virtual void Bar();
};
struct B : A {
  int y;
  virtual void Foo();
  virtual void Baz();
};

A a1, a2; B b1;
```



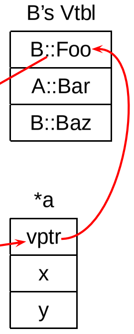
Virtual Functions

```
struct A {
  int x;
  virtual void Foo();
  virtual void Bar()
    { do_something(); }
};
struct B : A {
  int y;
  virtual void Foo();
  virtual void Baz();
};
A *a = new B;
a->Bar();
```



Virtual Functions

```
struct A {
  int x;
  virtual void Foo();
  virtual void Bar();
};
struct B : A {
  int y;
  virtual void Foo()
    { something_else(); }
  virtual void Baz();
};
A *a = new B;
a->Foo();
```



Multiple Inheritance

Rocket Science,
and nearly as dangerous

Inherit from two or more classes

```
class Window { ... };

class Border { ... };

class BWindow : public Window,
                public Border {
  ...
};
```

Multiple Inheritance Ambiguities

```
class Window {
  void draw();
};

class Border {
  void draw();    // OK
};

class BWindow : public Window,
                public Border { };

BWindow bw;
bw.draw();    // Compile-time error: ambiguous
```

Resolving Ambiguities Explicitly

```
class Window { void draw(); };

class Border { void draw(); };

class BWindow : public Window,
                public Border {
  void draw() { Window::draw(); }
};

BWindow bw;
bw.draw();    // OK
```

Duplicate Base Classes

A class may be inherited more than once

```
class Drawable { ... };
class Window : public Drawable { ... };
class Border : public Drawable { ... };
class BWindow : public Window, public
Border { ... };
```

BWindow gets two copies of the Drawable base class.

Virtual Base Classes

Virtual base classes are inherited at most once

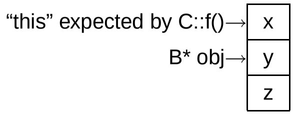
```
class Drawable { ... };
class Window : public virtual Drawable {
... };
class Border : public virtual Drawable {
... };
class BWindow : public Window, public
Border { ... };
```

BWindow gets one copy of the Drawable base class

Implementing Multiple Inheritance

A virtual function expects a pointer to its object

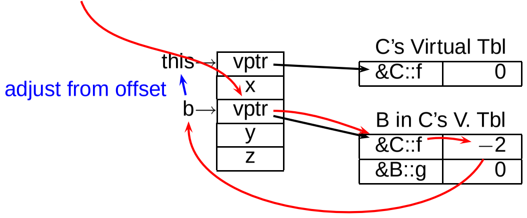
```
struct A { int x; virtual void f(); }
struct B { int y; virtual void f(); }
struct C : A, B { int z; void f(); }

B *b = new C;
b->f(); // Calls C::f()
```



“obj” is, by definition, a pointer to a B, not a C. Pointer must be adjusted depending on the actual type of the object. At least two ways to do this.

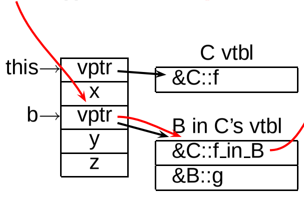
Implementation using Offsets

```
struct A { int x; virtual void f(); }
struct B { int y; virtual void f();
                  virtual void g(); }
struct C : A, B { int z; void f(); }
B *b = new C;
b->f(); // Call C::f()
```



Implementation using Thunks

```
struct A { int x; virtual void f(); }
struct B { int y; virtual void f();
                  virtual void g(); }
struct C : A, B { int z; void f(); }
B *b = new C;
b->f(); // Call C::f()
```



```
void
C::f_in_B(void *this)
{
  this = this - 2;
  goto C::f;
}
```

Offsets vs. Thunks

Offsets	Thunks
Offsets to virtual tables	Helper functions
Can be implemented in C	Needs “extra” semantics
All virtual functions cost more	Only multiply-inherited functions cost
Tricky	Very Tricky

Exceptions

A high-level replacement for C’s setjmp/longjmp.

```
struct Except { };

void baz() { throw Except; }
void bar() { baz(); }

void foo() {
  try {
    bar();
  } catch (Except e) {
    printf("oops");
  }
}
```

One Way to Implement Exceptions

```
try {                   push(Ex, Handler);

  throw Ex;             throw(Ex);
                        pop();
                        goto Exit;
} catch (Ex e) {      Handler:
  foo();                foo();
}                     Exit:
```

`push()` adds a handler to a stack

`pop()` removes a handler

`throw()` finds first matching handler

Problem: imposes overhead even with no exceptions

Implementing Exceptions Cleverly

Real question is the nearest handler for a given PC.

```
1 void foo() {        3. look in table
2
3   try {
4     bar();
5   } catch (Ex1 e) { H1: a(); }
6
7 } 2. H2 doesn't handle Ex1, reraise

8 void bar() {        1. look in table
9
10  try {
11    throw Ex1();
12  } catch (Ex2 e) { H2: b(); }
13
14 }
```

Lines	Action
1–2	Reraise
3–5	H1
6–9	Reraise
10–12	H2
13–14	Reraise