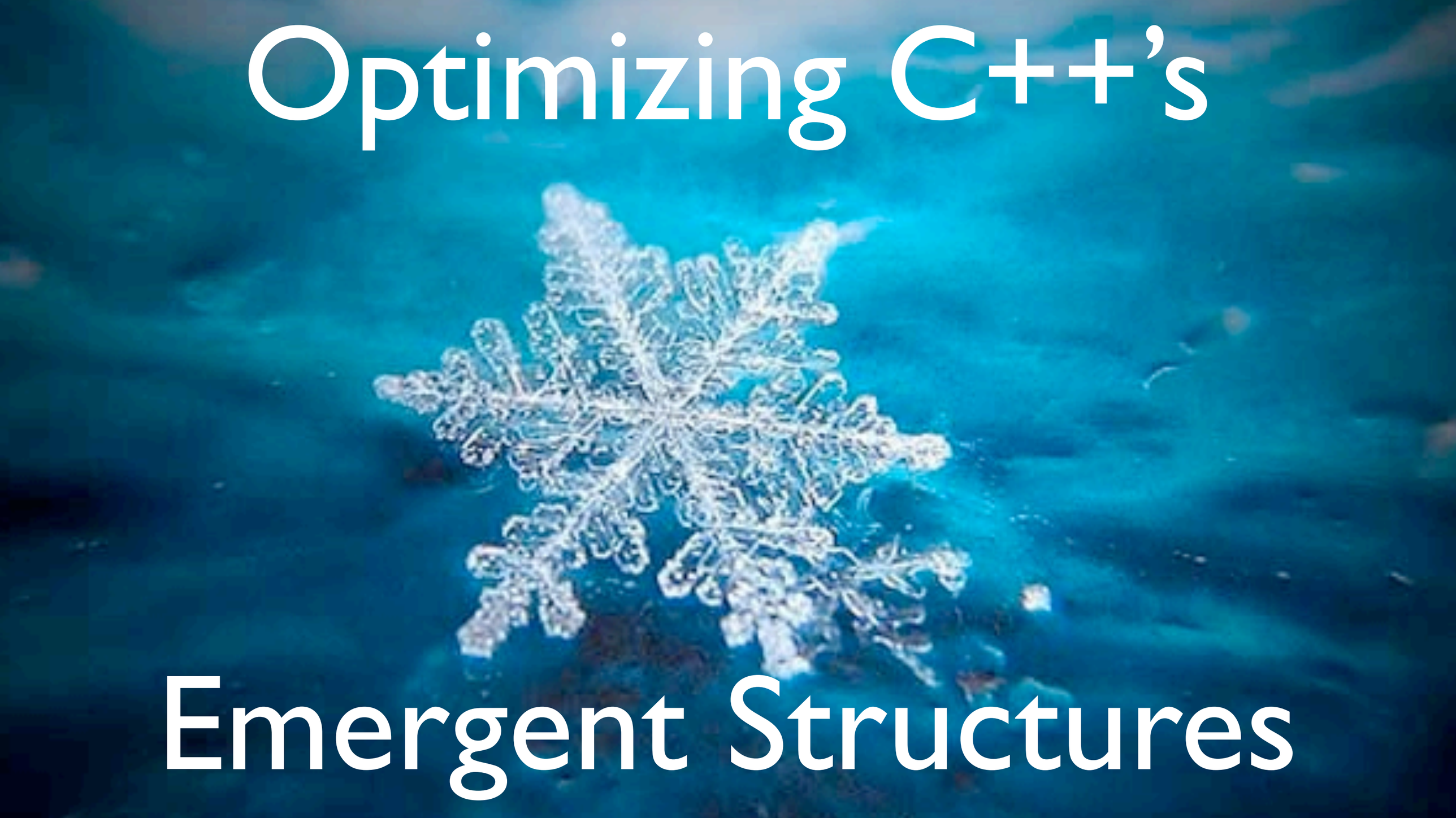
Optimizing C++'s

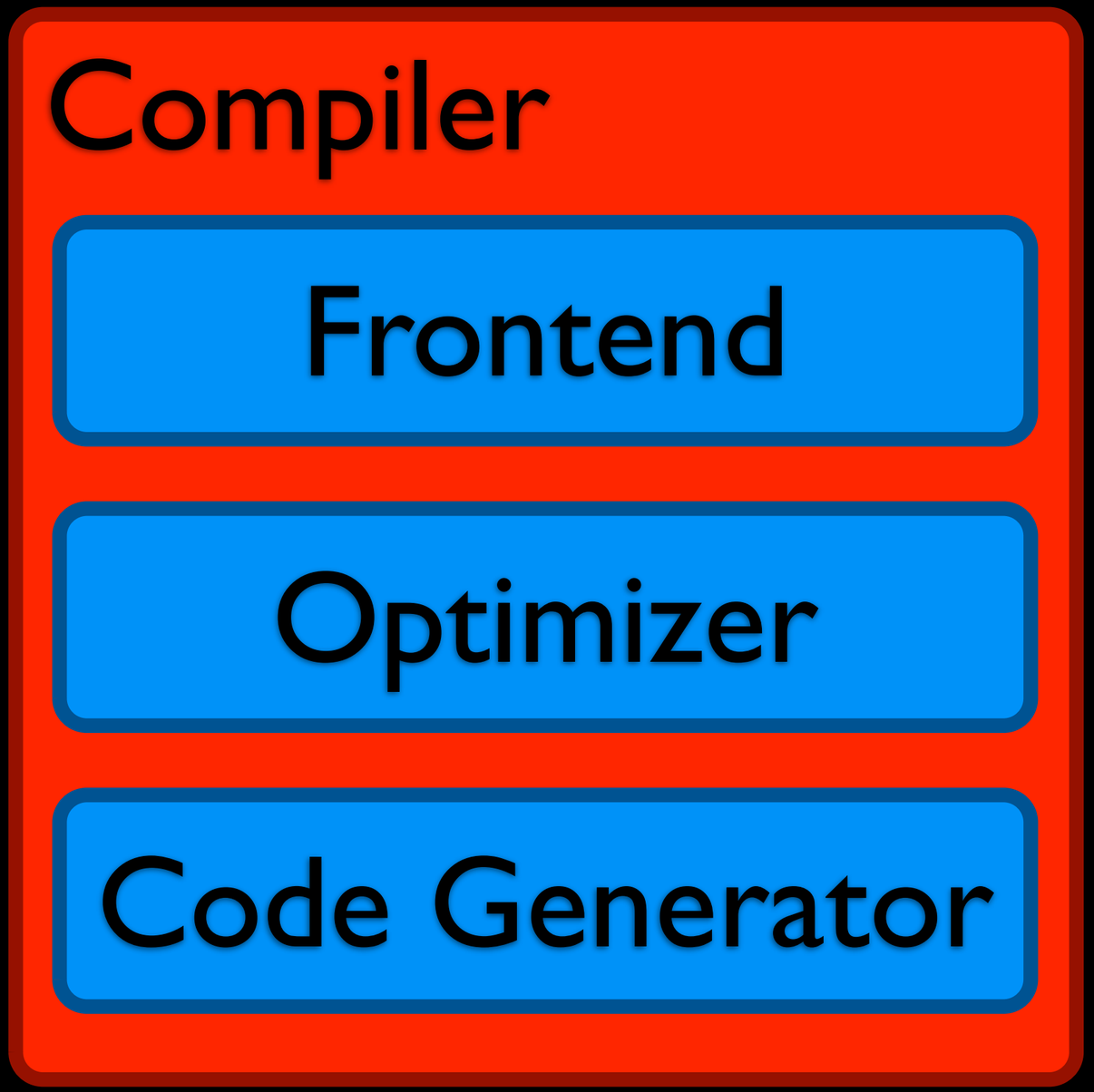


Emergent Structures

Emergent Structures

Compilers

Principles, Techniques, and Tools


Compiler
Frontend
Optimizer
Code Generator

Optimization 101

```
extern "C" int atoi(const char *);
int main(int argc, char **argv) {
  if (argc != 3)
    return -1;

  return atoi(argv[1]) + atoi(argv[2]);
}
```

```
TranslationUnitDecl 0x5332240 <<invalid sloc>>
|-LinkageSpecDecl 0x5332bf0 <hello_world.cpp:1:1, col:33> C
| `-FunctionDecl 0x5332d40 <col:12, col:33> atoi 'int (const char *)'
|   `-ParmVarDecl 0x5332c80 <col:21, col:33> 'const char *'
`-FunctionDecl 0x535e680 <line:2:1, line:7:1> main 'int (int, char **)'
  |-ParmVarDecl 0x5332e00 <line:2:10, col:14> argc 'int'
  |-ParmVarDecl 0x5332ed0 <col:20, col:27> argv 'char **'
  `-CompoundStmt 0x535ebb8 <col:33, line:7:1>
    |-IfStmt 0x535e818 <line:3:3, line:4:13>
    | |-<<<NULL>>>
    | |-BinaryOperator 0x535e790 <line:3:7, col:15> '_Bool' '!='
    | | |-ImplicitCastExpr 0x535e778 <col:7> 'int' <LValueToRValue>
    | | | `-DeclRefExpr 0x535e730 <col:7> 'int' lvalue ParmVar 0x5332e00 'argc' 'int'
    | | `-IntegerLiteral 0x535e758 <col:15> 'int' 3
    | |-ReturnStmt 0x535e7f8 <line:4:5, col:13>
    | | `-UnaryOperator 0x535e7d8 <col:12, col:13> 'int' prefix '-'
    | |   `-IntegerLiteral 0x535e7b8 <col:13> 'int' 1
    | `-<<<NULL>>>
    `-ReturnStmt 0x535eb98 <line:6:3, col:38>
      `-BinaryOperator 0x535eb70 <col:10, col:38> 'int' '+'
        |-CallExpr 0x535e990 <col:10, col:22> 'int'
        | |-ImplicitCastExpr 0x535e978 <col:10> 'int (*)(const char *)' <FunctionToPointerDecay>
        | | `-DeclRefExpr 0x535e928 <col:10> 'int (const char *)' lvalue Function 0x5332d40 'atoi' '...'
        | `-ImplicitCastExpr 0x535e9d8 <col:15, col:21> 'const char *' <NoOp>
        |   `-ImplicitCastExpr 0x535e9c0 <col:15, col:21> 'char *' <LValueToRValue>
        |     `-ArraySubscriptExpr 0x535e900 <col:15, col:21> 'char *' lvalue
        |       |-ImplicitCastExpr 0x535e8e8 <col:15> 'char **' <LValueToRValue>
        |       | `-DeclRefExpr 0x535e8a0 <col:15> 'char **' lvalue ParmVar 0x5332ed0 'argv' 'char **'
        |       `-IntegerLiteral 0x535e8c8 <col:20> 'int' 1
        `-CallExpr 0x535eb10 <col:26, col:38> 'int'
          |-ImplicitCastExpr 0x535eaf8 <col:26> 'int (*)(const char *)' <FunctionToPointerDecay>
          | `-DeclRefExpr 0x535ead0 <col:26> 'int (const char *)' lvalue Function 0x5332d40 'atoi' 'int (const
```

```
define i32 @main(i32 %argc, i8** %argv) {
entry:
  %retval = alloca i32, align 4
  %argc.addr = alloca i32, align 4
  %argv.addr = alloca i8**, align 8
  store i32 0, i32* %retval
  store i32 %argc, i32* %argc.addr, align 4
  store i8** %argv, i8*** %argv.addr, align 8
  %0 = load i32* %argc.addr, align 4
  %cmp = icmp ne i32 %0, 3
  br i1 %cmp, label %if.then, label %if.end

if.then:                                          ; preds = %entry
  store i32 -1, i32* %retval
  br label %return

if.end:                                           ; preds = %entry
  %1 = load i8*** %argv.addr, align 8
  %arrayidx = getelementptr inbounds i8** %1, i64 1
  %2 = load i8** %arrayidx, align 8
  %call = call i32 @atoi(i8* %2)
  %3 = load i8*** %argv.addr, align 8
  %arrayidx1 = getelementptr inbounds i8** %3, i64 2
  %4 = load i8** %arrayidx1, align 8
  %call2 = call i32 @atoi(i8* %4)
  %add = add nsw i32 %call, %call2
  store i32 %add, i32* %retval
  br label %return

return:                                           ; preds = %if.end, %if.then
  %5 = load i32* %retval
  ret i32 %5
}
```

A brief digression to describe LLVM's IR...

```
declare i32 @g(i32 %x)

define i32 @f(i32 %a, i32 %b) {
entry:
  %c = add i32 %a, %b
  %d = call i32 @g(i32 %c)
  %e = add i32 %c, %d
  ret i32 %e
}
```

```
declare i32 @g(i32 %x)

define i32 @f(i32 %a, i32 %b, i1 %flag) {
entry:
  %c = add i32 %a, %b
  br i1 %flag, label %then, label %else

then:
  %d = call i32 @g(i32 %c)
  ret i32 %d

else:
  ret i32 %c
}
```

```
declare i32 @g(i32 %x)

define i32 @f(i32 %a, i32 %b, i1 %flag) {
entry:
  %c = add i32 %a, %b
  br i1 %flag, label %then, label %end

then:
  %d = call i32 @g(i32 %c)
  br label %end

end:
  %result = phi i32 [ %entry, %c ],
                    [ %then, %d ]
  ret i32 %result
}
```

Ok, where were we...
IR for hello world

```
define i32 @main(i32 %argc, i8** %argv) {
entry:
  %retval = alloca i32, align 4
  %argc.addr = alloca i32, align 4
  %argv.addr = alloca i8**, align 8
  store i32 0, i32* %retval
  store i32 %argc, i32* %argc.addr, align 4
  store i8** %argv, i8*** %argv.addr, align 8
  %0 = load i32* %argc.addr, align 4
  %cmp = icmp ne i32 %0, 3
  br i1 %cmp, label %if.then, label %if.end

if.then:                                          ; preds = %entry
  store i32 -1, i32* %retval
  br label %return

if.end:                                           ; preds = %entry
  %1 = load i8*** %argv.addr, align 8
  %arrayidx = getelementptr inbounds i8** %1, i64 1
  %2 = load i8** %arrayidx, align 8
  %call = call i32 @atoi(i8* %2)
  %3 = load i8*** %argv.addr, align 8
  %arrayidx1 = getelementptr inbounds i8** %3, i64 2
  %4 = load i8** %arrayidx1, align 8
  %call2 = call i32 @atoi(i8* %4)
  %add = add nsw i32 %call, %call2
  store i32 %add, i32* %retval
  br label %return

return:                                           ; preds = %if.end, %if.then
  %5 = load i32* %retval
  ret i32 %5
}
```

Optimization does more than just make code faster...

Step 1: Cleanup

```c
extern "C" int atoi(const char *);
int main(int argc, char **argv) {
  if (argc != 3)
    return -1;

  return atoi(argv[1]) +
         atoi(argv[2]);
}
```

```llvm
define i32 @main(i32 %argc, i8** %argv) {
entry:
  %retval = alloca i32, align 4
  %argc.addr = alloca i32, align 4
  %argv.addr = alloca i8**, align 8
  store i32 0, i32* %retval
  store i32 %argc, i32* %argc.addr, align 4
  store i8** %argv, i8*** %argv.addr, align 8
  %0 = load i32* %argc.addr, align 4
  %cmp = icmp ne i32 %0, 3
  br i1 %cmp, label %if.then, label %if.end

if.then:
  store i32 -1, i32* %retval
  br label %return

if.end:
  %1 = load i8*** %argv.addr, align 8
  %arrayidx = getelementptr i8** %1, i64 1
  %2 = load i8** %arrayidx, align 8
  %call = call i32 @atoi(i8* %2)
  %3 = load i8*** %argv.addr, align 8
  %arrayidx1 = getelementptr i8** %3, i64 2
  %4 = load i8** %arrayidx1, align 8
  %call2 = call i32 @atoi(i8* %4)
  %add = add nsw i32 %call, %call2
  store i32 %add, i32* %retval
  br label %return

return:
  %5 = load i32* %retval
  ret i32 %5
}
```

```
extern "C" int atoi(const char *);
int main(int argc, char **argv) {
  if (argc != 3)
    return -1;

  return atoi(argv[1]) +
         atoi(argv[2]);
}
```

```
define i32 @main(i32 %argc, i8** %argv) {
entry:
  %cmp = icmp ne i32 %argc, 3
  br i1 %cmp, label %if.then, label %if.end

if.then:
  br label %return

if.end:
  %arrayidx = getelementptr i8** %argv, i64 1
  %arrayval = load i8** %arrayidx, align 8
  %call = call i32 @atoi(i8* %arrayval)
  %arrayidx1 = getelementptr i8** %argv, i64 2
  %arrayval1 = load i8** %arrayidx1, align 8
  %call2 = call i32 @atoi(i8* %arrayval1)
  %add = add nsw i32 %call, %call2
  br label %return

return:
  %retval.0 = phi i32 [ -1, %if.then ],
                      [ %add, %if.end ]
  ret i32 %retval.0
}
```

Step 2: Canonicalization

```
int x = y;
if (!flag)
  x = z;
```

```
int x;
if (flag)
  x = y;
else
  x = z;
```

```
if (flag)
  z = y;
int x = z;
```

```
int x = flag? y : z;
```

```
extern "C" int atoi(const char *);
int main(int argc, char **argv) {
  if (argc != 3)
    return -1;

  return atoi(argv[1]) +
         atoi(argv[2]);
}
```

```
define i32 @main(i32 %argc, i8** %argv) {
entry:
  %cmp = icmp ne i32 %argc, 3
  br i1 %cmp, label %if.then, label %if.end

if.then:
  br label %return

if.end:
  %arrayidx = getelementptr i8** %argv, i64 1
  %arrayval = load i8** %arrayidx, align 8
  %call = call i32 @atoi(i8* %arrayval)
  %arrayidx1 = getelementptr i8** %argv, i64 2
  %arrayval1 = load i8** %arrayidx1, align 8
  %call2 = call i32 @atoi(i8* %arrayval1)
  %add = add nsw i32 %call, %call2
  br label %return

return:
  %retval.0 = phi i32 [ -1, %if.then ],
                      [ %add, %if.end ]
  ret i32 %retval.0
}
```

```c
extern "C" int atoi(const char *);
int main(int argc, char **argv) {
  if (argc != 3)
    return -1;

  return atoi(argv[1]) +
         atoi(argv[2]);
}
```

```llvm
define i32 @main(i32 %argc, i8** %argv) {
entry:
  %cmp = icmp eq i32 %argc, 3
  br i1 %cmp, label %if.end, label %return

if.end:
  %arrayidx = getelementptr i8** %argv, i64 1
  %arrayval = load i8** %arrayidx, align 8
  %call = call i32 @atoi(i8* %arrayval)
  %arrayidx1 = getelementptr i8** %argv, i64 2
  %arrayval1 = load i8** %arrayidx1, align 8
  %call2 = call i32 @atoi(i8* %arrayval1)
  %add = add nsw i32 %call2, %call
  br label %return

return:
  %retval.0 = phi i32 [ %add, %if.end ],
                      [ -1, %entry ]
  ret i32 %retval.0
}
```

Step 3: Collapse Abstractions

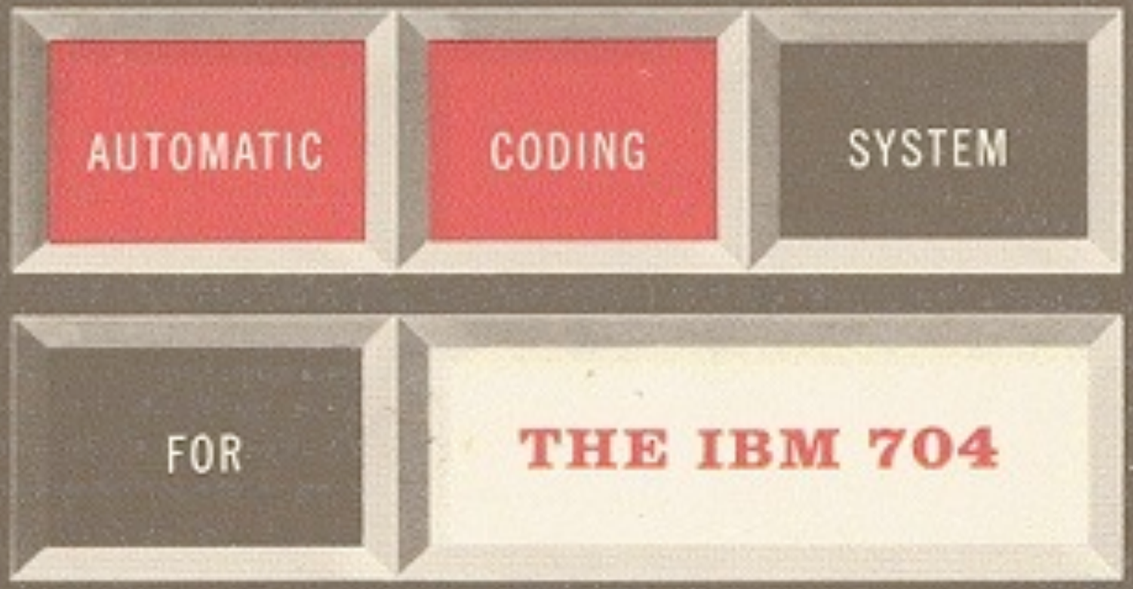
Three key abstractions:

1. Functions, calls, and the call graph.
2. Memory, loads, and stores.
3. Loops.

But no one really cares about loops in C++... Right?

Fortran

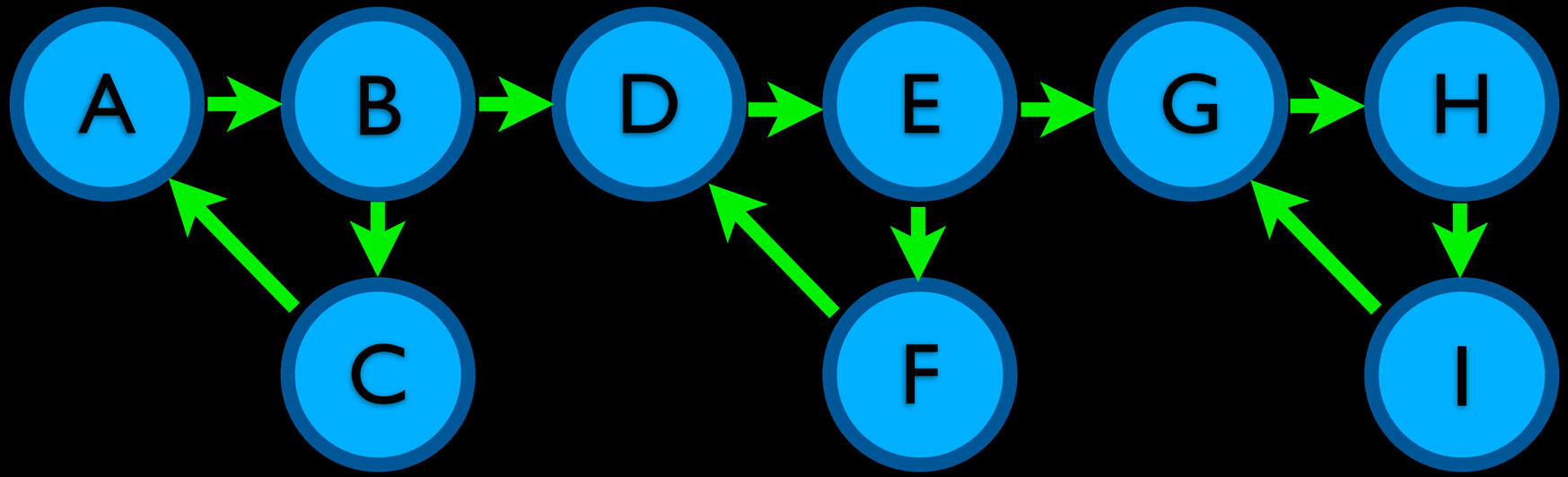
Because
if you really care...

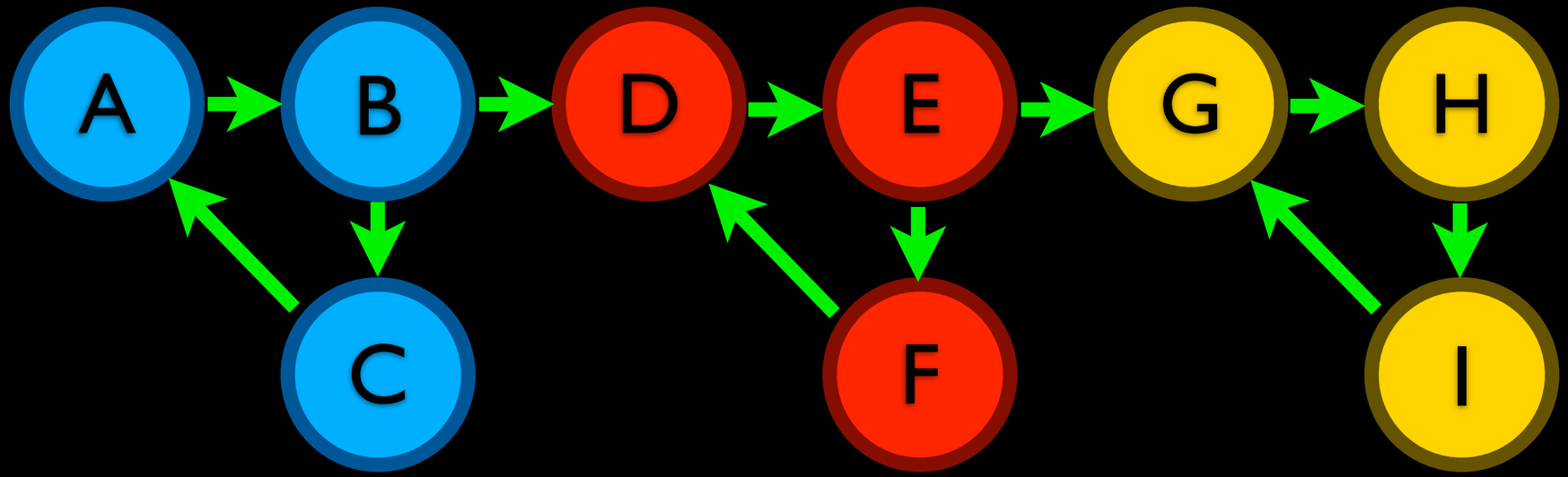


Also, we know how to optimize loops

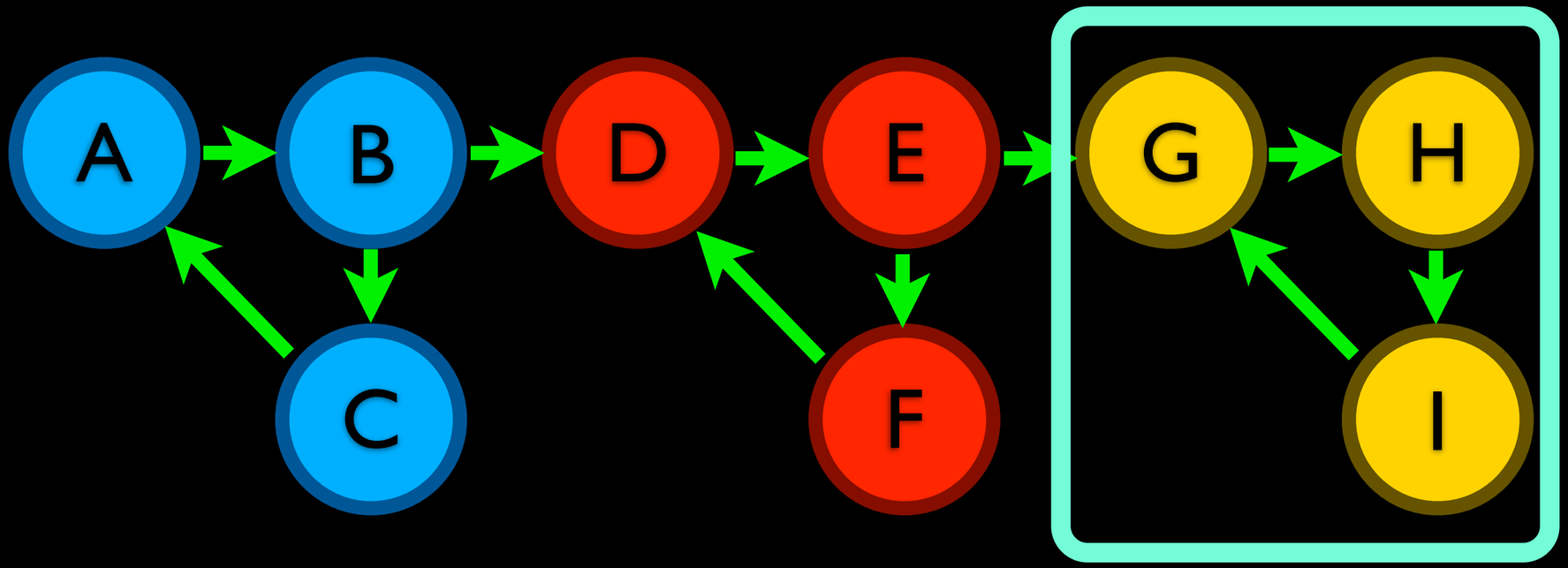
What about the other fundamentals?

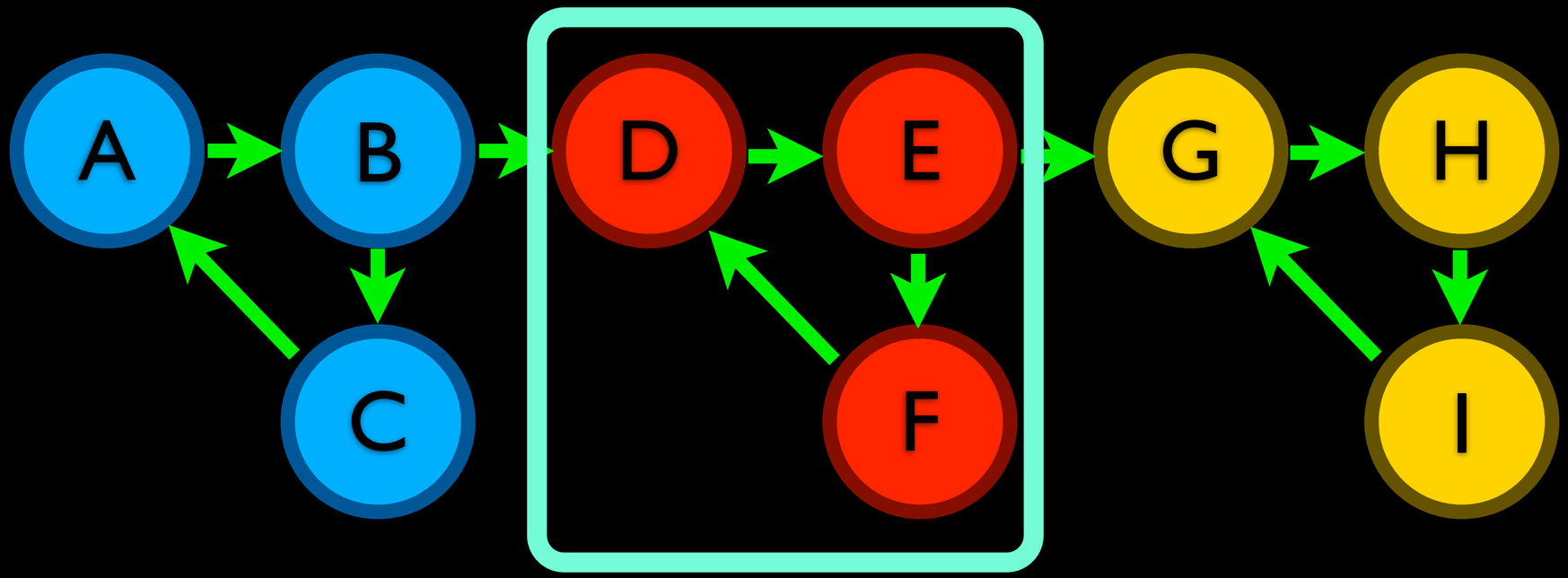
Let’s look at collapsing function calls

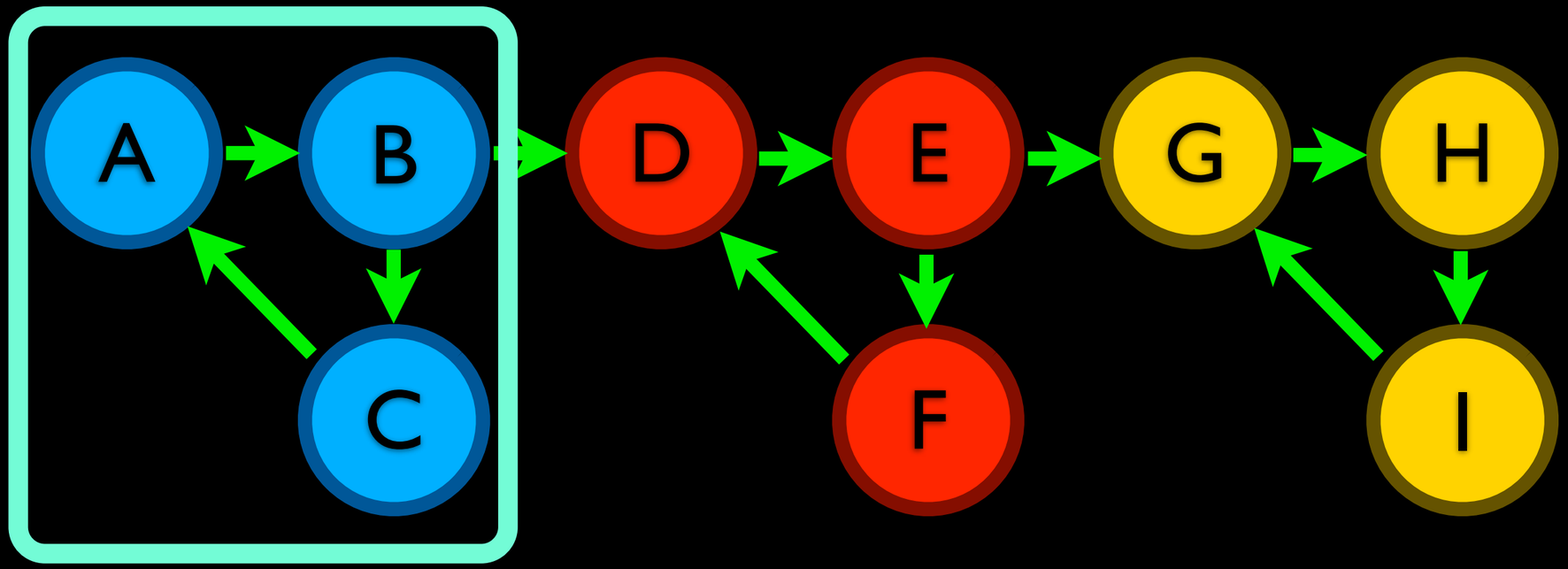

A
B
D
E
G
H
C
F
I


A
B
D
E
G
H
C
F
I

Bottom-up SCC-based call graph walk


A
B
C
D
E
F
G
H
I


A
B
C
D
E
F
G
H
I


A
B
C
D
E
F
G
H
I

How does the optimizer evaluate complexity?

```
int g(double x, double y, double z);

int f(struct S* s, double y, double x) {
  return g(x, y, s->z);
}
```

```
void fancy_sort(vector<int> &v) {
  if (v.size() <= 1)
    return;
  if (v.size() == 2) {
    if (v.front() >= v.back())
      swap(v.front(), v.back());
    return;
  }
  std::sort(v.begin(), v.end());
}
```

This doesn’t always work though!

```
int hash(hash_state &h) {
  // Some complex code on 'h'
  return /* final value */;
}

template <typename T, typename ...Ts>
int hash(hash_state &h, T arg, Ts ...args) {
  // Complex code to put 'arg'
  // into the 'h' state...
  return hash(h, args...);
}
```

Let’s look at memory, loads, and stores...

```
declare i32 @g(i32 %x)

define i32 @f(i32 %a, i32 %b,
              i1 %flag) {
entry:
  %mem = alloca i32
  %c = add i32 %a, %b
  store i32 %c, i32* %mem
  br i1 %flag, label %then,
               label %end

then:
  %d = call i32 @g(i32 %c)
  store i32 %d, i32* %mem
  br label %end

end:
  %result = load i32* %mem
  ret i32 %result
}
```

```
declare i32 @g(i32 %x)

define i32 @f(i32 %a, i32 %b,
              i1 %flag) {
entry:
  %c = add i32 %a, %b
  br i1 %flag, label %then,
               label %end

then:
  %d = call i32 @g(i32 %c)
  br label %end

end:
  %result = phi i32 [ %entry, %c ],
                    [ %then, %d ]
  ret i32 %result
}
```

```
%S = type { i32, i32, i32 }
declare i32 @g(i32 %x)
define i32 @f(i32 %a, i32 %b, i1 %flag) {
entry:
  %mem = alloca %S
  %c = add i32 %a, %b
  %addr0 = getelementptr %S* %mem,
                         i32 0, i32 0
  store i32 %c, i32* %addr0
  %addr1 = getelementptr %S* %mem,
                         i32 0, i32 1
  store i32 0, i32* %addr1
  %addr2 = getelementptr %S* %mem,
                         i32 0, i32 2
  store i32 0, i32* %addr2
  br i1 %flag, label %then, label %end

then:
  %d = call i32 @g(i32 %c)
  store i32 %d, i32* %addr1
  %e = call i32 @g(i32 %a)
  store i32 %e, i32* %addr2
  br label %end

end:
  %val0 = load i32* %addr0
  %val1 = load i32* %addr1
  %val2 = load i32* %addr2
  %f = add i32 %val0, %val1
  %result = add i32 %f, %val2
  ret i32 %result
}
```

```
declare i32 @g(i32)

define i32 @f(i32 %a, i32 %b, i1 %flag) {
entry:
  %c = add i32 %a, %b
  br i1 %flag, label %then, label %end

then:
  %d = call i32 @g(i32 %c)
  %e = call i32 @g(i32 %a)
  br label %end

end:
  %mem.sroa.1.0 =
      phi i32 [ %d, %then ], [ 0, %entry ]
  %mem.sroa.2.0 =
      phi i32 [ %e, %then ], [ 0, %entry ]
  %f = add i32 %c, %mem.sroa.1.0
  %result = add i32 %f, %mem.sroa.2.0
  ret i32 %result
}
```

Not all memory is non-escaping stack addresses... But that's OK

What happens when these abstractions are *combined*?

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

void g(int a, int b, int &c) {
  c = a * b;
}
```

```
struct S {
  float x, y, z;
  double delta;

  double compute();
};
```

```
double f() {
  S s;
  s.x = /* expensive compute */;
  s.y = /* expensive compute */;
  s.z = /* expensive compute */;
  s.delta = s.x - s.y - s.z;
  return s.compute();
}
```

Some tips for optimizable APIs:

- Use value semantics! (Already a good idea...)
- Don’t create unneeded abstractions. Sometimes, a function parameter is plenty.
- Partition **all** logic away from template-expanded deeply nested constructs.

Use abstractions, but also consider how they will look to the optimizer.

Questions!
(maybe answers?)

C++Now
May 2013

Chandler Carruth
chandlerc@gmail.com