Tutorial 5: Example Optimization: Dead Code Elimination

ECE 466/566 Spring 2025

March 26, 2025
Due: ~Two weeks

If you need clarification of any text, add comments directly to this document.

1. Objective

- Write code for iterating over all the functions in a Module.
- Write code for iterating over all the basic blocks and instructions within a function.
- Test if an instruction has any uses, and iterate over the uses of an instruction
- Find the definition of an operand.
- Iterate over the operands of an instruction.
- Describe the purpose of cast<>, isa<>, and dyn_cast<>
- Evaluate if an instruction meets a set of criteria for optimization.
- Traverse the operands for an instruction.
- Removing instructions consistent with Dead Code Elimination.

2. Overall Structure of an Optimization

From this point on, we'll use clang to build IR for us--that way we can compile arbitrary C code into IR and then run it through the optimizations we implement. So, when we write an optimization pass, we'll assume that we get a bitcode file as input and produce a bitcode file as output. To implement it, we'll just use a simple command line structure like this:

```
./dce input.bc output.bc
```

This is pretty easy to achieve using the same main function from previous examples but modified to load a module rather than build an empty one:

C

```
#include <unistd.h>
#include "llvm-c/Core.h"
#include "llvm-c/BitReader.h"
#include "llvm-c/BitWriter.h"
#include "stats.h"
```

```
#include "worklist.h"

LLVMStatisticsRef Dead;
LLVMStatisticsRef WorkList;

int
main (int argc, char ** argv)
{
  /* some basic error handling */
  if (argc < 3) {
    fprintf(stderr,"Not enough files specified.\n");
    return 0;
  }

  LLVMMemoryBufferRef Buff=NULL;
  char *outMessage=NULL;

/* 1. Read contents of object file from command line argv[1]
*/
LLVMCreateMemoryBufferWithContentsOfFile(argv[1],&Buff,&outMessa
ge);

  LLVMModuleRef Module=NULL;
  /* 2. Try to parse buffer into a legal Module */
  if (!Buff || LLVMParseBitcode(Buff,&Module,&outMessage))
    {
      /* handle error */
      printf("Error opening file: %s\n",outMessage);
      return 1;
    }
  LLVMEnableStatistics();

  // Perform memory to register promotion
  LLVMPassManagerRef PM = LLVMCreatePassManager();
  LLVMAddScalarReplAggregatesPass(PM);
  LLVMRunPassManager(PM, Module);
  Dead = LLVMStatisticsCreate("Dead", "Dead instructions");
  WorkList = LLVMStatisticsCreate("WorkList", "Instructions
added to worklist");

  /* 3. Do optimization on Module */
```

```
  /* 4. Save result to a new file */
  char *msg;
  LLVMBool res = LLVMVerifyModule(Module,
LLVMPrintMessageAction,&msg);
  if (!res)
    LLVMWriteBitcodeToFile(Module,argv[2]);
  else
    fprintf(stderr,"Error: %s not created.\n",argv[2]);
  LLVMPrintStatistics();
  return 0;
}
```

C++

```
#include <stdlib.h>
#include <stdio.h>

#include "llvm/IR/LLVMContext.h"
#include "llvm/IR/Value.h"
#include "llvm/IR/Function.h"
#include "llvm/IR/Type.h"
#include "llvm/IR/IRBuilder.h"
#include "llvm/IR/Verifier.h"

#include "llvm/ADT/Statistic.h"

#include "llvm/Bitcode/BitcodeReader.h"
#include "llvm/Bitcode/BitcodeWriter.h"
#include "llvm/Support/SystemUtils.h"
#include "llvm/Support/ToolOutputFile.h"
#include "llvm/Support/FileSystem.h"

#include "llvm/IRReader/IRReader.h"
#include "llvm/Support/SourceMgr.h"
#include "llvm/Support/CommandLine.h"

using namespace llvm;

static llvm::Statistic DeadInst = {"", "Dead", "DCE found dead
instructions"};
static llvm::Statistic WorkList = {"", "WorkList", "Added to
work list"};


static cl::opt<std::string>
        InputFilename(cl::Positional, cl::desc("<input
bitcode>"), cl::Required, cl::init("-"));
```

```
static cl::opt<std::string>
        OutputFilename(cl::Positional, cl::desc("<output
bitcode>"), cl::Required, cl::init("out.bc"));

static cl::opt<bool>
        Mem2Reg("mem2reg",
                cl::desc("Perform memory to register promotion
before CSE."),
                cl::init(false));

static cl::opt<bool>
        Verbose("verbose",
                cl::desc("Verbosely print lots of status
messages to the screen."),
                cl::init(false));


int main (int argc, char ** argv)
{
  cl::ParseCommandLineOptions(argc, argv, "./dce <input>
<output> \n");
  // LLVM idiom for constructing output file.
  std::unique_ptr<ToolOutputFile> Out;
  std::string ErrorInfo;
  std::error_code EC;
  Out.reset(new ToolOutputFile(OutputFile, EC,
                      sys::fs::OF_None));

  SMDiagnostic Err;
  std::unique_ptr<Module> M;
  LLVMContext *Context = new LLVMContext();
  M = parseIRFile(InputFilename, Err, *Context);

  if (M.get() == 0) {
    Err.print(argv[0], errs());
    return 1;
  }

  EnableStatistics();

  if (Mem2Reg) {
      if (Verbose)
          errs() << "Run Mem2Reg.\n";
      legacy::PassManager Passes;
      Passes.add(createPromoteMemoryToRegisterPass());
      Passes.run(*M.get());
```

```
  }
  if (Verbose)
      M->print(errs(), nullptr);

  /* 3. Do optimization on Module */


  M->print(errs(),nullptr);

  bool res = verifyModule(*M, &errs());
  if (!res) {
    WriteBitcodeToFile(*M.get(),Out->os());
    Out->keep();
  } else {
    fprintf(stderr,"Error: %s not created.\n",argv[2]);
  }

  PrintStatistics(errs());
  return 0;

}
```

Now, we need to add our own optimization. It will take an LLVMModuleRef as an argument and make changes to it accordingly.

C++

```
void NoOptimization(Module &M)
{
   // Do nothing! Simplest optimization that exists
}
```

C

```
void NoOptimization(LLVMModuleRef Module)
{
   // Do nothing! Simplest optimization that exists
}
```

There is little reason to implement such a trivial optimization except for instructional purposes. But, we will use such a function to implement most of our optimizations this semester. Then, we modify main to call our optimization just after comment #3:

C++	/* 3. Do optimization on Module */ NoOptimization(*M.get());
C	/* 3. Do optimization on Module */ NoOptimization(Module);

Now, you can compile this code. Make sure you put the definition of NoOptimization before main. Use Cmake to compile your code. After you compile it, run this code by passing in a bitcode file as an argument. For example, write a very simple test file called test0.c (already available in the repo), like this:

```
#include <stdio.h>

struct X {
  int x;
  int y;
  double j;
};

int main() {
  struct X x;
  x.x=5;
  x.y=10;
  int z;

  x.x=25;
  if (x.y>100)
    x.x -= 2;

  z = x.x;

  printf("%d %d\n",x.x,x.y);

  return 0;
}
```

Then compile it using clang.

To make assembly, add -c -S -emit-llvm:

```
clang -Xclang -disable-O0-optnone -S -c -emit-llvm -o test0.ll
test0.c
```

To make bitcode, omit the -S:

```
clang -Xclang -disable-O0-optnone -c -emit-llvm -o test0.bc test0.c
```

We need to add "`-Xclang -disable-O0-optnone`" so that the opt tool will optimize the code for us. Alright now you can test the tool and look at the output:

```
./dce test0.bc test0-dce.bc
llvm-dis test0-dce.bc
less test0-dce.ll
```

Our optimizer makes no changes to the IR, so the output should be identical to the input.

3. Iterating Over All Instructions

We'll use several APIs that help us traverse all of the instructions in a Module. First, remember the structure of a Module. A Module is a list of functions. A function is an ordered list of basic blocks. And a basic block is an ordered list of instructions. So, what we need is a nested loop that iterates over all of the functions, and for each function all of the blocks, and for each block all of the instructions.

In C++, we'll use iterator objects to visit the functions in a Module, the basic blocks in a Function, and the instructions in a basic block. In C, we'll use a few function calls to achieve the same effect as an iterator.

On the Module object, we can visit all functions:

C	`for (LLVMValueRef F = LLVMGetFirstFunction(Module);`
	`  F!=NULL;`
	`  F=LLVMGetNextFunction(F))`
	`{`
	`  // Use each function, F`
	`}`

C++	`for(Module::iterator i = M.begin(); i!=M.end(); i++)`
	`{`
	`  Function &F = *i;`
	`  // or get a pointer to the function:`
	`  Function *F_ptr = &*i;`
	`}`

For a function object, we can visit all basic blocks, either forward or reverse:

C	`for(LLVMBasicBlockRef BB=LLVMGetFirstBasicBlock(F);`
	`  BB!=NULL;`
	`  BB = LLVMGetNextBasicBlock(BB))`
	`{`
	`  // get each basic block in F`
	`}`

C++	`for(Function::iterator j = f.begin(); j != f.end(); j++)`
	`{`
	`  BasicBlock *bb = &*j;`
	`}`

For a basic block, we can get all of the instructions:

C	`for (LLVMValueRef I = LLVMGetFirstInstruction(BB);`
	`  I != NULL;`
	`  I = LLVMGetNextInstruction(I))`
	`{`
	`  // loop over all instructions`
	`}`

C++	`for(BasicBlock::iterator k = bb->begin(); k != bb->end(); k++)`
	`{`
	`  Instruction *Inst = &*k;`
	`}`

Often, when we write an optimization, we'll have a loop that nests these three loops together:

C	`void NoOptimization(LLVMModuleRef Module)`
	`{`
	`  // Loop over all the functions`
	`  LLVMValueRef F=NULL;`
	
	`  for(F=LLVMGetFirstFunction(Module);`
	`  F!=NULL;`
	`  F=LLVMGetNextFunction(F))`
	`  {`
	`  // Is this function defined?`

```
         if (LLVMCountBasicBlocks(F))
         {
            LLVMBasicBlockRef BB;
            for(BB=LLVMGetFirstBasicBlock(F);
                BB!=NULL;
               BB=LLVMGetNextBasicBlock(BB))
               {
                 LLVMValueRef I;
                 for (I=LLVMGetFirstInstruction(BB);
                      I!=NULL;
                      I=LLVMGetNextInstruction(I))
                      {
                          // Do something
                      }
               }
         }
     }
}
```

C++

```
void NoOptimization(Module &M)
{
 for(Module::iterator f = M.begin(); f!=M.end(); f++)
 {
   for(Function::iterator bb=(*f).begin();bb!=(*f).end();bb++)
   {
     for(BasicBlock::iterator I = (*bb).begin(); I !=
(*bb).end(); I++)
       {

       }
   }
 }
}

// Or, more idiomatic for C++, use auto to make it more
concise

    for(auto f = M.begin(); f!=M.end(); f++)
    {
      // loop over functions
      for(auto bb= f->begin(); bb!=f->end(); bb++)
     {
       // loop over basic blocks
       for(auto i = bb->begin(); i != bb->end(); i++)
         {
```

```
            //loop over instructions
          }
        }
      }
```

This may look like a complicated loop, but it's really pretty simple when you think about the overall structure of the Module. All we're saying is that we want to look at every instruction once to consider it as a possible dead code.

4. Finding Dead Instructions (video)

So, back to the optimization. Figuring out if an instruction is dead.

A rule of thumb when writing software is to keep each function relatively easy to explain, and usually that's easier to achieve with a maximum loop nesting of depth around 3. Since we've already gotten to nesting depth of three, let's put the functionality for testing if an instruction is dead in another function.

We'll just add some code like this to our innermost for loop above:

C++
```
if (isDead(*I)) {
    //add I to a worklist to replace later
    printf("Found an instruction to delete!\n");
}
```

C
```
if (isDead(I)) {
    //add I to a worklist to replace later
    printf("Found an instruction to delete!\n");
}
```

Now, we implement the function isDead. Here's the skeleton version:

C++
```
bool isDead(Instruction &I)
{
    /*
        Check necessary requirements, otherwise return false
     */
    return 0;
}
```

C	`int isDead(LLVMValueRef I)`
	`{`
	`  /*`
	`  Check necessary requirements, otherwise return false`
	`  */`
	`  return 0;`
	`}`

For Dead Code Elimination, we can remove any instruction that has no uses and does **not** produce a side effect. Side effects include branches, stores, volatile memory operations, or special memory operations needed for parallel programs.

Here's the condition to see if it has uses:

C++	`// Are there uses, if not then dead!`
	`if ( I.use_begin() == I.use_end() )`
	`  {`
	`  return true; // dead, but this is not enough`
	`  }`

C	`// Are there uses, if not then dead!`
	`if (LLVMGetFirstUse(I)==NULL)`
	`  return 1;`

Now, check if the [instruction is of the correct kind] to be replaced. The C and C++ code is a little bit different, but the rule is that the C opcode enum, say LLVMAbc, becomes Instruction::Abc:

C	`LLVMOpcode opcode = LLVMGetInstructionOpcode(I);`
	`switch(opcode) {`
	`// when in doubt, keep it! add opcode here to remove:`
	`case LLVMFNeg:`
	`case LLVMAdd:`
	`case LLVMFAdd:`
	`case LLVMSub:`
	`case LLVMFSub:`
	`case LLVMMul:`
	`case LLVMFMul:`
	`case LLVMUDiv:`
	`case LLVMSDiv:`
	`case LLVMFDiv:`
	`case LLVMURem:`

```
  case LLVMSRem:
  case LLVMFRem:
  case LLVMShl:
  case LLVMLShr:
  case LLVMAShr:
  case LLVMAnd:
  case LLVMOr:
  case LLVMXor:
  case LLVMAlloca:
  case LLVMGetElementPtr:
  case LLVMTrunc:
  case LLVMZExt:
  case LLVMSExt:
  case LLVMFPToUI:
  case LLVMFPToSI:
  case LLVMUIToFP:
  case LLVMSIToFP:
  case LLVMFPTrunc:
  case LLVMFPExt:
  case LLVMPtrToInt:
  case LLVMIntToPtr:
  case LLVMBitCast:
  case LLVMAddrSpaceCast:
  case LLVMICmp:
  case LLVMFCmp:
  case LLVMPHI:
  case LLVMSelect:
  case LLVMExtractElement:
  case LLVMInsertElement:
  case LLVMShuffleVector:
  case LLVMExtractValue:
  case LLVMInsertValue:
    // Success!
    return 1;

  case LLVMLoad: if(!LLVMGetVolatile(I)) return 1; // Success

  // all others must be kept
  default:
    break;
  }
```

C++

```
      int opcode = I.getOpcode();
        switch(opcode){
        case Instruction::Add:
        case Instruction::FNeg:
        case Instruction::FAdd:
        case Instruction::Sub:
        case Instruction::FSub:
        case Instruction::Mul:
        case Instruction::FMul:
        case Instruction::UDiv:
        case Instruction::SDiv:
        case Instruction::FDiv:
        case Instruction::URem:
        case Instruction::SRem:
        case Instruction::FRem:
        case Instruction::Shl:
        case Instruction::LShr:
        case Instruction::AShr:
        case Instruction::And:
        case Instruction::Or:
        case Instruction::Xor:
        case Instruction::Alloca:
        case Instruction::GetElementPtr:
        case Instruction::Trunc:
        case Instruction::ZExt:
        case Instruction::SExt:
        case Instruction::FPToUI:
        case Instruction::FPToSI:
        case Instruction::UIToFP:
        case Instruction::SIToFP:
        case Instruction::FPTrunc:
        case Instruction::FPExt:
        case Instruction::PtrToInt:
        case Instruction::IntToPtr:
        case Instruction::BitCast:
        case Instruction::AddrSpaceCast:
        case Instruction::ICmp:
        case Instruction::FCmp:
        case Instruction::PHI:
        case Instruction::Select:
        case Instruction::ExtractElement:
        case Instruction::InsertElement:
```

```
        case Instruction::ShuffleVector:
        case Instruction::ExtractValue:
        case Instruction::InsertValue:
          if ( I.use_begin() == I.use_end() )
          {
              return true;
          }
          break;

        case Instruction::Load:
          {
            LoadInst *li = dyn_cast<LoadInst>(&I);
            if (li && li->isVolatile())
              return false;
            if (I.use_begin() == I.use_end())
               return true;
            break;

          }

        default:
          // any other opcode fails
          return false;
        }
```

5. Removing Dead Instructions (video)

All instructions that pass these tests can be eliminated. It would be tempting to implement code like this:

C

```
LLVMValueRef I;
for (I=LLVMGetFirstInstruction(BB);
     I!=NULL;
     I=LLVMGetNextInstruction(I))
{
  if (isDead(I))
     LLVMInstructionEraseFromParent(I);
}
```

C++

```
for(BasicBlock::iterator bit = BB.begin();
    bit != BB.end();
```

```
    bit++)
{
   Instruction &I = *bit;
   if(isDead(&I)) {
     I = I.eraseFromParent(); // update iterator while
erasing

   }
}
```

In principle, it's what we want to do.

Warning: When we delete the instruction, we have to be careful not to leave the iterator pointing at a deleted instruction. There are many ways to avoid that problem. In C++, we can just capture a new iterator as the return value from `eraseFromParent`.

Because it serves our interests later on, we'll use a worklist to remember that it's dead and we'll remove it later. At the start of the function, we'll create a worklist:

C	`//#include "worklist.h" - file instructor provided` `worklist_t worklist = worklist_create();`
C++	`// Use standard template library for a set` `std::set<Instruction*> worklist;`

Then, we'll insert all dead instructions into the worklist:

C	`if (isDead(I))` `  worklist_insert(worklist,I);`
C++	`if(isDead(*I))` `  worklist.insert(&*I); // must insert a pointer`

Note, instead of forward iterating over all instructions, we reverse iterate to make DCE work more efficiently.

After we've visited all instructions, we'll go back and delete the dead ones and while we're at it we'll count how many we delete:

C

```
// While there's an instruction in the worklist
while(!worklist_empty(worklist)) {
    // take one instruction out of the list
    LLVMValueRef I = worklist_pop(worklist);
    LLVMStatisticsInc(Dead);
    // Erase it from its parent basic block
    LLVMInstructionEraseFromParent(I);
}
```

C++

```
while(worklist.size()>0)
{
   // Get the first item
   Instruction *i = *(worklist.begin());
   // Erase it from worklist
   worklist.erase(i);

   // Erase from basic block
   i->eraseFromParent();
   DeadInst++;
}
```

This will delete the instructions in the worklist, but we're not done yet.

When we delete an instruction, it will remove some uses of another instruction. Of these instructions, there must be an operand of the one we remove. So, we loop over all operands and reconsider them as being dead.

C

```
while(!worklist_empty(worklist))
{
   LLVMValueRef I = worklist_pop(worklist);

   // Check if instruction is dead because it may have been
   // added in if-statement below and we did not check to see
   // if it was dead first
   if (isDead(I))
   {
      // Loop over operands of I
      for(unsigned i=0; i<LLVMGetNumOperands(I); i++)
      {
```

```
          LLVMValueRef J = LLVMGetOperand(I,i);

          // Add to worklist only if J is an instruction
          // Note, J still has one use so the isDead routine
          // would return false, don't check that yet.
          // This forces us to check in the if statement
above.

         // The operand could be many different things, in
         // particular constants. Don't try to delete
         // unless its an instruction:
          if (LLVMIsAInstruction(J))
             worklist_insert(worklist,J);
      }
      LLVMStatisticsInc(Dead);
      LLVMInstructionEraseFromParent(I);
   }
}
```

C++

```
while(worklist.size()>0)
{
   Instruction* i = *worklist.begin();
   worklist.erase(i);
   if(isDead(*i))
   {
      for(unsigned op=0; op<i->getNumOperands(); op++)
      {
          // Note, op still has one use so the isDead routine
          // would return false, so don't check that yet.
          // This forces us to check in the if statement
above.

         // The operand could be many different things, in
         // particular constants. Don't try to delete it
         // unless its an instruction:
         if ( isa<Instruction>(i->getOperand(op)) )
         {
            Instruction *o =
                 cast<Instruction>(i->getOperand(op));
            worklist.insert(o);
            WorkList++;
          }
      }
      i->eraseFromParent();
```

```
        DeadInst++;
    }
}
```

Test your implementation

Run your optimization on some sample C programs. Write some test cases, compile them with clang, and run them through your tool. Add a print statement to main so you can see how many instructions you removed using the DCE_count variable.

Also, running it on a wolfbench would be good practice. To do that, follow these steps, assuming you are in the `tutorial/5/C++` or `tutorial/5/C` directory:

- `mkdir dce-test`
- `cd dce-test`
- `/ece566/wolfbench/configure --enable-customtool=/absolute/path/to/dce`
- `make OPTFLAGS="-mem2reg" test compare`

That last make command runs DCE on all of the workloads after performing `mem2reg`. It will also compare the output from running the program to make sure it produces the correct result.

6. Questions

1. What is the basic structure of an optimization pass? What are the advantages and disadvantages?
   The basic structure of an optimization pass consists of module loading, followed by optimization logic and verification, concluding with writing an output. Some advantages of an optimization pass are reduction in the size of the code, improvement in execution time, and removal of unnecessary code. Some disadvantages of an optimization pass are increased risk of removing important sections of code/code instructions and the need for complex dataflow analysis.
2. How can you iterate over all the functions in a Module? all the basic blocks in a function? all the instructions in a basic block?
   You can iterate over all the functions in a Module, all the basic blocks in a function, and all the instructions in a basic block by implementing the “for” loops that are present in the dce.cpp file. The specific section is below:

```
for(Module::iterator i = M.begin(); i!=M.end(); i++)
{
  Function &F = *i;
//or get a pointer to the function:

for(Function::iterator j = F.begin(); j != F.end(); j++) {
BasicBlock &BB = *j;
```

//BB.dump();

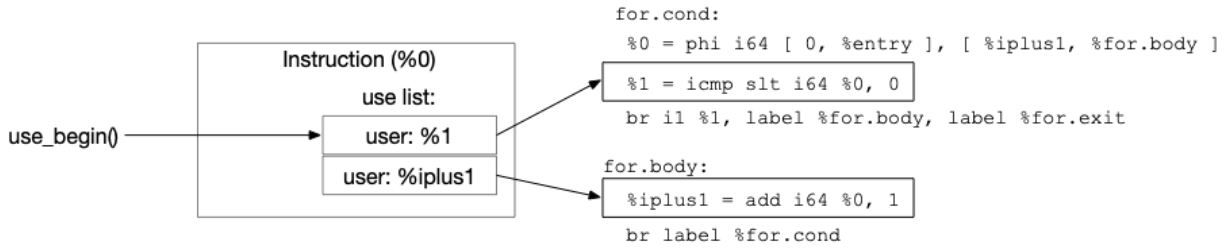
for(BasicBlock::iterator k = BB.begin(); k != BB.end(); k++) {
Instruction &I = *k;

3. Why do we use a worklist when removing instructions?
   A worklist is used when removing instructions because it does both the collection of initially dead instructions, and checks the status/checks a specific operand after the dead instructions are deleted. This also prevents cascading of dead code (used AI for that last part).
4. How many instructions does your pass remove? What if you run mem2reg first?
   The instructions the specific pass removes is 7, since there are 7 dead code instructions actually removed. If you run mem2reg first, the output will be zero because the allocas are initially deleted and then converted into registers (used AI for this part).

Appendix

A. For A Definition, Get all Uses (aka Def-use) (video)

SSA form makes calculating uses and definitions a bit easier. Since LLVM requires SSA form, it also has built in functionality for iterating over all uses of a definition. Each Instruction object has a built-in *use iterator* for traversing the uses of the instruction. The use iterator returns an llvm::Use object. From a particular use, we can ask for the user, which returns the Value object that's using it. Here's a picture to clear up how this works:



For example, here is code that will print all the uses of phi_i:

C++	`errs() << "All users of the instruction: \n";`
	`using use_iterator = Value::use_iterator;`
	`for(use_iterator u = i->use_begin(); u!=i->use_end(); u++)`
	`  {`
	`  Value *v = u->getUser();`
	`  v->print(errs(),true);`
	`  errs() << "\n";`
	`  }`

C	LLVMUseRef use;
	fprintf(stderr,"All users of the instruction: ");
	LLVMDumpValue(i);
	for(use = LLVMGetFirstUse(i);
	use!=NULL;
	use = LLVMGetNextUse(use))
	{
	LLVMValueRef user = LLVMGetUser(use);
	LLVMDumpValue(user);
	}

Make sure you put this code after all of the building code, otherwise it won't find all uses. Compile and run the code. What you should see is this:

```
All users of the instruction (%2 = alloca %struct.X, align 8):
  %15 = getelementptr inbounds %struct.X, %struct.X* %2, i32 0, i32 0
  %11 = getelementptr inbounds %struct.X, %struct.X* %2, i32 0, i32 0
  %7 = getelementptr inbounds %struct.X, %struct.X* %2, i32 0, i32 1
  %6 = getelementptr inbounds %struct.X, %struct.X* %2, i32 0, i32 0
  %5 = getelementptr inbounds %struct.X, %struct.X* %2, i32 0, i32 1
  %4 = getelementptr inbounds %struct.X, %struct.X* %2, i32 0, i32 0
```

The two instructions that use %i are shown. Note, that both of these instructions logically follow the phi in execution--they happen later chronologically---and this makes sense!

More details about the interface:

C	Anywhere the result of an instruction is used is called a Use, and this is represented by the LLVMUseRef type. Given an LLVMUseRef, we can either ask for the value being used (LLVMGetUsedValue) or the user (LLVMGetUser). LLVMUseRef LLVMGetFirstUse (LLVMValueRef Val); LLVMUseRef LLVMGetNextUse (LLVMUseRef U); LLVMValueRef LLVMGetUser (LLVMUseRef U); LLVMValueRef LLVMGetUsedValue (LLVMUseRef U);

C++	//Each Instruction object
	// using the Value::use_iterator;
	Instruction::use_iterator Instruction::use_begin();
	Instruction::use_iterator Instruction::use_end();
	Value* Use::getUser();

B. For A Use, Get its Definition (aka Use-def)

Sometimes we have a use and we want to know where it comes from. In SSA form, we know that each register can be defined only once. So, the problem of finding the definition is fairly straightforward. Let's consider the case of the add instruction in our example above. We can loop over all the operands in the add instruction and print their definitions:

C

```
for(unsigned op=0; op< LLVMGetNumOperands(i); op++) {
  LLVMValueRef definition = LLVMGetOperand(i,op);
  fprintf(stderr,"Definition of op=%d is:",op);
  LLVMDumpValue(definition);
}
```

C++

```
// For C++ boffins, you should prefer LLVM's cast<> over the
C++
// standard static_cast<> to get the extra LLVM specific
checks.

for(unsigned op=0; op < inst->getNumOperands(); op++)
{
  Value* def = inst->getOperand(op);
  errs() << "  Definition of op=" << op << " is:" ;
  def->print(errs(),true);
  errs() << "\n";
}
```

We get output like this:

```
Definition of op=0 is:  %i = phi i64 [ 0, %entry ], [ %iplus1,
%for.body ]
Definition of op=1 is:  i64 1
```

Note, that when we get the reference to the operand, *we are getting the definition*. Also, note that the definition is early in the code---it happens before the add chronologically---as we would expect!

Here's a brief reference to the functions we just used and their return values and parameters. For C++, I've also indicated their home class.

C++

All Instructions inherit from the User object. So these functions are available on any instruction object we create or find:

```
Value* User::getOperand(unsigned index);
```

	`Use& User::getOperandUse(unsigned index);`
	`void User::setOperand(unsigned index, Value* val);`
	`unsigned User::getNumOperands();`

C	`LLVMValueRef LLVMGetOperand(LLVMValueRef val, unsigned Index);`
	`LLVMUseRef LLVMGetOperandUse(LLVMValueRef val, unsigned Index);`
	`void LLVMSetOperand(LLVMValueRef user, unsigned Index, LLVMValueRef val);`
	`int LLVMGetNumOperands(LLVMValueRef val);`

C. Determining Object Types/Kinds in LLVM (video)

When building objects and traversing objects in LLVM API, we often want to know the kind of object we have found. For example, is an operand pointing to an instruction or a constant value?

We can test what type we have using the isa<> template, and we can cast between types using the cast<> and dyn_cast<> templates. For example, if we are iterating over operands and want to know which one is an Instruction, we can do this:

C++

```
for(unsigned op=0; op < add_inst->getNumOperands(); op++)
{
  Value* def = inst->getOperand(op);
  errs() << "Definition of op=" << op << " is:\n" ;
  def->print(errs(),true);

  if ( isa<Instruction>(def) ) {
    errs() << " (This is an instruction!) \n" ;
  } else {
    errs() << "\n";
  }
}
```

C

```
for(unsigned op=0; op< LLVMGetNumOperands(i); op++) {
  LLVMValueRef definition = LLVMGetOperand(i,op);
  fprintf(stderr,"Definition of op=%d is:\n",op);
  LLVMDumpValue(definition);
  if(LLVMIsAInstruction(definition)) {
     fprintf(stderr,"Is an instruction!\n");
```

```
    }
}
```

Note, the isa<> template allows us to ask if the argument object is of the type passed as the template specialization.

The C API also has Is-A capabilities through a suite of a LLVMIsA#### functions. The common rule is that after the IsA you place the C++ name of the object, for example: LLVMIsAUser() or LLVMIsALoadInst(). This provides access to the isa<> functionality. You can see a full list of the C APIs LLVMIsA functions here starting at line number 1519.

If it is an instruction, we may want to cast it to an Instruction* type. We do that with cast<>:

```
Instruction * op_inst = cast<Instruction>(add_inst->getOperand(op));
```

If we were wrong and it wasn't really an Instruction or the argument is nullptr, the cast<> template causes an assertion failure and the program will abort. So, only use cast<> when you've already checked to make sure it's non-null and the conversion is legal.

If you don't know if the conversion is legal, but you want to try anyway, use dyn_cast<> but always follow it with a nullptr check. dyn_cast<> will return nullptr if the conversion is not allowed.

```
Instruction * op_inst =
     dyn_cast<Instruction>(add_inst->getOperand(op));
if (op_inst != nullptr) {
  // do something
}
```

Finally, there are also versions of these functions that will handle null values gracefully: cast_or_null<> and dyn_cast_or_null<>. These may be helpful at times.

For more information about the cast<> operations and how they work, they are fully documented in the LLVM Language Reference Manual in the section on isa, dyn_cast, and cast templates.

Note, the C API lacks a rich casting API because most of the C API is based on LLVMValueRef. So, there is no need to cast between objects. However, this also exposes a limitation of the C API -- namely, we often do not have access to the full API of the internal C++ objects.

D. Helpful LLVM API

Iterating over a Module in C

LLVMValueRef	**LLVMGetFirstFunction** (**LLVMModuleRef** M)
LLVMValueRef	**LLVMGetLastFunction** (**LLVMModuleRef** M)
LLVMValueRef	**LLVMGetNextFunction** (**LLVMValueRef** Fn)

Iterating over a Function in C

unsigned	**LLVMCountBasicBlocks** (**LLVMValueRef** Fn)
void	**LLVMGetBasicBlocks** (**LLVMValueRef** Fn, **LLVMBasicBlockRef** *BasicBlocks)
LLVMBasicBlockRef	**LLVMGetFirstBasicBlock** (**LLVMValueRef** Fn)
LLVMBasicBlockRef	**LLVMGetLastBasicBlock** (**LLVMValueRef** Fn)
LLVMBasicBlockRef	**LLVMGetNextBasicBlock** (**LLVMBasicBlockRef** BB)
LLVMBasicBlockRef	**LLVMGetPreviousBasicBlock** (**LLVMBasicBlockRef** BB)
LLVMBasicBlockRef	**LLVMGetEntryBasicBlock** (**LLVMValueRef** Fn)

Iterating over Instructions in C

LLVMValueRef	**LLVMGetNextInstruction** (**LLVMValueRef** Inst)

LLVMValueRef	**LLVMGetFirstInstruction** (**LLVMBasicBlockRef** BB)
LLVMValueRef	**LLVMGetLastInstruction** (**LLVMBasicBlockRef** BB)