# Some extra Cruft

# Containers

- LLVM types and STL types are used, depends on need
- LLVM types (SmallVector, etc) have certain properties that make them better than STL for certain use cases
- Once know algorithm:
  - select abstract container you need to satisfy that,
  - then select LLVM or STL depending on details of memory usage.
- See **&&&** for info on various containers

# Common Helper Patterns

- C++11, C++14 support.. auto, smart pointers etc
- dyn_cast<T>(V): try to cast V to type T, else ret NULL
- isa<T>(V): true if V is of type T
- dump() — Value's have them.. useful
- outs(), errs() — See raw_ostream.h
- CommandLine library to easily add commands **&&&**

raw_ostream API:

- outs()
- errs()
- nulls()

- Use of *auto*
- F(B(I,…), B(I,…))

```
bool
FPSkel::runOnFunction(Function &F)
{
  unsigned nbb = 0;
  unsigned ins = 0;
  if (F.isDeclaration()) {
    errs() << "Ignoring function declaration.\n";
    return false;
  }
  if (F.hasName()) {
    errs() << "\nFunction: " << F.getName() << "\n";
  } else {
    errs() << "\nFunction: not named\n";
  }
  for (auto &B : F) { // Iterate through Basic Blocks in a function
    ++nbb;
    errs() << "  Basic Block found:\n";
    B.dump();
    for (auto &I : B) { // Iterate through instructions in the block
      ++ins;
    }
    errs() << "  --- end of basic block ---\n";
  }
  errs() << "  Total of " << nbb << " blocks in this function\n";
  errs() << "  Total of " << ins << " instructions in this function\n";
  errs() << "---  end of function  ---\n";

  // return true if CFG has changed.
  return false;
}
static RegisterPass<FPSkel> XX("fpskel", "Function Pass Skeleton");
```

** This is from the *fpskel* code

# Instructions from Fn w/o BasicBlock

```
bool
SomeModulePass::runOnModule(Module &M)
{
...
  for (auto &f : M) {
    for (auto ii = inst_begin(f); ii != inst_end(f); ++ii) {
      Instruction *in = &*ii;
      if (isa<CallInst>(in) || isa<InvokeInst>(in)) {
        outs() << "found call site\n";
      }
      if (CallInst *CI = dyn_cast<CallInst>)(in)) {
        Function *cf = CI->getCalledFunction();
        cf->dump();
      } else if (LoadInst *LI = dyn_cast<LoadInst>(in)) {
...
      } else if (StoreInst *SI = dyn_cast<StoreInst>(in)) {
...
      } else if (BranchInst *BI = dyn_cast<BranchInst>(in)) {
..
      }.....
```

# Or, use Instruction Visitor

```
struct CheckSomeInsts : public InstVisitor<CheckSomeInsts> {
  void
  visitCallInst(CallInst &CI)
  {
    CI.dump();
  }
  void
  visitInvokeInst(InvokeInst &II)
  {
    II.dump();
  }
};
bool
runOnModule(Module &M)
{
  for (auto &f : M) {
    CheckSomeInsts m;
    m.visit(f);
  }
  return false;
}
```