```
// 15-745 S14 Assignment 1: FunctionInfo.cpp
// Group:
// aebtekar  Aram Ebtekar
// auc       Alejandro Carbonara
//////////////////////////////////////////////////////////////////////////////

#include "llvm/Pass.h"
#include "llvm/IR/Function.h"
#include "llvm/Support/raw_ostream.h"
#include "llvm/IR/Module.h"
#include "llvm/IR/Instructions.h"

#include <ostream>
#include <fstream>
#include <iostream>

using namespace llvm;

namespace
{

class FunctionInfo : public ModulePass
{
  // Output the function information to standard out.
  void printFunctionInfo(Module& M)
  {
    std::cout << "Module " << M.getModuleIdentifier().c_str() << std::endl;
    std::cout << "Name,\tArgs,\tCalls,\tBlocks,\tInsns\n";

    for (Module::iterator MI = M.begin(), ME = M.end(); MI != ME; ++MI)
    {
      Function& F = *MI;
      // Initialize the info for F
      bool is_var_arg = F.isVarArg();
      size_t arg_count = F.arg_size();
      size_t callsite_count = 0;
      size_t block_count = F.size();
      size_t instruction_count = 0;

      // Count instructions
      for (ilist_iterator<BasicBlock> BI = F.begin(), BE = F.end(); BI != BE; ++BI)
      {
        instruction_count += BI->size();
      }
      // Count call sites across the whole module
      for (Module::iterator MI2 = M.begin(); MI2 != ME; ++MI2)
      for (ilist_iterator<BasicBlock> BI = MI2->begin(), BE = MI2->end(); BI != BE;
 ++BI)
      for (ilist_iterator<Instruction> II = BI->begin(), IE = BI->end(); II != IE;
++II)
      {
        CallInst* call = dyn_cast<CallInst>(II);
        if (call != NULL && call->getCalledFunction() == &F)
        {
          ++callsite_count;
        }
      }

      // Print the info for F
      std::cout << F.getName().data() << ",\t";
      if (is_var_arg)
        std::cout << "*,";
      else
        std::cout << arg_count << ",\t";

      std::cout << callsite_count << ",\t" << block_count << ",\t"
          << instruction_count << std::endl;
    }
  }

public:

  static char ID;

  FunctionInfo() : ModulePass(ID) { }

  ~FunctionInfo() { }

  // We don't modify the program, so we preserve all analyses
  virtual void getAnalysisUsage(AnalysisUsage &AU) const
  {
    AU.setPreservesAll();
  }

  virtual bool runOnFunction(Function& F)
  {

  }

  virtual bool runOnModule(Module& M)
  {
    for (Module::iterator MI = M.begin(), ME = M.end(); MI != ME; ++MI)
    {
      runOnFunction(*MI);
    }
    printFunctionInfo(M);
    return false;
  }
};

// LLVM uses the address of this static member to identify the pass, so the
// initialization value is unimportant.
char FunctionInfo::ID = 0;
RegisterPass<FunctionInfo> X("function-info", "15745: Function Information");

}
```