



**UNIMORE**  
UNIVERSITÀ DEGLI STUDI DI  
MODENA E REGGIO EMILIA

Dipartimento di Scienze Fisiche,  
Informatiche e Matematiche

# **3. LAB 1: Introduzione ai *passi* LLVM**

## **Linguaggi e Compilatori** [I215-011]

*Corso di Laurea in INFORMATICA*  
(D.M.270/04) [16-262]  
Anno accademico 2023/2024

**Prof. Andrea Marongiu**  
[andrea.marongiu@unimore.it](mailto:andrea.marongiu@unimore.it)

# Copyright note

*È vietata la copia e la riproduzione dei contenuti e immagini in qualsiasi forma.*

*È inoltre vietata la redistribuzione e la pubblicazione dei contenuti e immagini non autorizzata espressamente dall'autore o dall'Università di Modena e Reggio Emilia.*

# Credits

- Cooper, Torczon, “Engineering a Compiler”, Elsevier
- Aho, Lam, Sethi, Ullman, “Compilatori: principi, tecniche e strumenti – seconda edizione”, Pearson
- Gibbons, Carnegie Mellon University, “Optimizing Compilers”
- Pekhimenko, University of Toronto, “Compiler Optimization”



**UNIMORE**  
UNIVERSITÀ DEGLI STUDI DI  
MODENA E REGGIO EMILIA











Dipartimento di Scienze Fisiche,  
Informatiche e Matematiche

# INSTALLAZIONE LLVM 17

# Installazione LLVM 17.0.6

- Il codice sorgente è scaricabile al link

<https://github.com/llvm/llvm-project/releases/tag/llvmorg-17.0.6>

 <a href="#">polly-17.0.6.src.tar.xz</a>	8.79 MB	Nov 28, 2023
 <a href="#">polly-17.0.6.src.tar.xz.sig</a>	438 Bytes	Nov 28, 2023
 <a href="#">runtimes-17.0.6.src.tar.xz</a>	6.09 KB	Nov 28, 2023
 <a href="#">runtimes-17.0.6.src.tar.xz.sig</a>	438 Bytes	Nov 28, 2023
 <a href="#">test-suite-17.0.6.src.tar.xz</a>	158 MB	Nov 28, 2023
 <a href="#">test-suite-17.0.6.src.tar.xz.sig</a>	438 Bytes	Nov 28, 2023
 <a href="#">third-party-17.0.6.src.tar.xz</a>	375 KB	Nov 28, 2023
 <a href="#">third-party-17.0.6.src.tar.xz.sig</a>	438 Bytes	Nov 28, 2023
 <a href="#">Source code (zip)</a>		Nov 28, 2023
 <a href="#">Source code (tar.gz)</a>		Nov 28, 2023

**In fondo alla pagina**



# Installazione LLVM 17.0.6


- Creare nella vostra ROOT directory la seguente struttura

```
amarongiu@kernighan:~/workspace/LLVM_17$ ll
total 28
drwxr-xr-x  6 amarongiu compilers 4096 Mar  6 18:04 ./
drwxr-x---  6 amarongiu compilers 4096 Mar  3 12:17 ../
drwxr-xr-x 19 amarongiu compilers 4096 Mar  7 15:59 BUILD/
drwxr-xr-x  7 amarongiu compilers 4096 Mar  4 12:49 INSTALL/
-rw-r--r--  1 amarongiu compilers  88 Mar  6 18:04 setup.sh
drwxr-xr-x  3 amarongiu compilers 4096 Mar  3 12:27 SRC/
drwxr-xr-x  3 amarongiu compilers 4096 Mar  6 18:00 TEST/
```

# Installazione LLVM 17.0.6

- Il file zip/tar appena scaricato va copiato in **SRC**, e lì scompattato


```
amarongiu@kernighan:~/workspace/LLVM_17$ ll
total 28
drwxr-xr-x  6 amarongiu compilers 4096 Mar  6 18:04 ./
drwxr-x---  6 amarongiu compilers 4096 Mar  3 12:17 ../
drwxr-xr-x 19 amarongiu compilers 4096 Mar  7 15:59 BUILD/
drwxr-xr-x  7 amarongiu compilers 4096 Mar  4 12:49 INSTALL/
-rw-r--r--  1 amarongiu compilers  88 Mar  6 18:04 setup.sh
drwxr-xr-x  3 amarongiu compilers 4096 Mar  3 12:27 SRC/
drwxr-xr-x  3 amarongiu compilers 4096 Mar  6 18:00 TEST/
```



# Installazione LLVM 17.0.6

- La cartella **BUILD** conterrà il prodotto della compilazione dei sorgenti
  - e delle nostre modifiche

```
amarongiu@kernighan:~/workspace/LLVM_17$ ll
total 28
drwxr-xr-x  6 amarongiu compilers 4096 Mar  6 18:04 ./
drwxr-x---  6 amarongiu compilers 4096 Mar  3 12:17 ../
drwxr-xr-x 19 amarongiu compilers 4096 Mar  7 15:59 BUILD/
drwxr-xr-x  7 amarongiu compilers 4096 Mar  4 12:49 INSTALL/
-rw-r--r--  1 amarongiu compilers   88 Mar  6 18:04 setup.sh
drwxr-xr-x  3 amarongiu compilers 4096 Mar  3 12:27 SRC/
drwxr-xr-x  3 amarongiu compilers 4096 Mar  6 18:00 TEST/
```






# Installazione LLVM 17.0.6

- La cartella **INSTALL** conterrà i binari dei vari *tools*
  - clang, opt, llc, ...
- E le librerie, gli include files, etc.


```
amarongiu@kernighan:~/workspace/LLVM_17$ ll
total 28
drwxr-xr-x  6 amarongiu compilers 4096 Mar  6 18:04 ./
drwxr-x---  6 amarongiu compilers 4096 Mar  3 12:17 ../
drwxr-xr-x 19 amarongiu compilers 4096 Mar  7 15:59 BUILD/
drwxr-xr-x  7 amarongiu compilers 4096 Mar  4 12:49 INSTALL/
-rw-r--r--  1 amarongiu compilers   88 Mar  6 18:04 setup.sh
drwxr-xr-x  3 amarongiu compilers 4096 Mar  3 12:27 SRC/
drwxr-xr-x  3 amarongiu compilers 4096 Mar  6 18:00 TEST/
```



# Installazione LLVM 17.0.6

- La cartella **TEST** conterrà le nostre esercitazioni
  - Ovvero i programmi che saranno oggetto delle nostre ottimizzazioni/analisi


```
amarongiu@kernighan:~/workspace/LLVM_17$ ll
total 28
drwxr-xr-x  6 amarongiu compilers 4096 Mar  6 18:04 ./
drwxr-x---  6 amarongiu compilers 4096 Mar  3 12:17 ../
drwxr-xr-x 19 amarongiu compilers 4096 Mar  7 15:59 BUILD/
drwxr-xr-x  7 amarongiu compilers 4096 Mar  4 12:49 INSTALL/
-rw-r--r--  1 amarongiu compilers  88 Mar  6 18:04 setup.sh
drwxr-xr-x  3 amarongiu compilers 4096 Mar  3 12:27 SRC/
drwxr-xr-x  3 amarongiu compilers 4096 Mar  6 18:00 TEST/
```



# Installazione LLVM 17.0.6

- Potete crearvi un semplice *bash* script (`setup.sh`) per esportare la cartella dei tools `INSTALL/bin`
  - `export PATH=$ROOT/INSTALL/bin:$PATH`

```
amarongiu@kernighan:~/workspace/LLVM_17$ ll
total 28
drwxr-xr-x  6 amarongiu compilers 4096 Mar  6 18:04 ./
drwxr-x---  6 amarongiu compilers 4096 Mar  3 12:17 ../
drwxr-xr-x 19 amarongiu compilers 4096 Mar  7 15:59 BUILD/
drwxr-xr-x  7 amarongiu compilers 4096 Mar  4 12:49 INSTALL/
-rw-r--r--  1 amarongiu compilers   88 Mar  6 18:04 setup.sh
drwxr-xr-x  3 amarongiu compilers 4096 Mar  3 12:27 SRC/
drwxr-xr-x  3 amarongiu compilers 4096 Mar  6 18:00 TEST/
```



- Così facendo la vostra installazione può convivere con altre versioni di LLVM installate nel sistema

# Installazione LLVM 17.0.6

- Potete crearvi un semplice *bash* script (`setup.sh`)

per esempio:

- esempio

```
$ which opt
ROOT/INSTALL/bin/opt
$
```

in

PATH

```
amarongiu@
total 28
drwxr-xr-x
drwxr-xr-x
drwxr-xr-x
drwxr-xr-x
-rw-r--r-
drwxr-xr-x
drwxr-xr-x
```

- Così facendo la vostra installazione può convivere con altre versioni di LLVM installate nel sistema

# Installazione LLVM 17.0.6

- L'installazione della toolchain prevede tre steps:
  - Configurazione
  - Compilazione
  - Installazione
- Possiamo fare riferimento alla documentazione ufficiale
  - <https://llvm.org/docs/GettingStarted.html#getting-started-with-llvm>

# Configurazione

- `cd ROOT/BUILD`
- `cmake -G "Unix Makefiles"`
  - `-DCMAKE_BUILD_TYPE=Release`
  - `-DCMAKE_INSTALL_PREFIX=$ROOT/INSTALL`
  - `-DLLVM_ENABLE_PROJECTS="clang"`
  - `-DLLVM_TARGETS_TO_BUILD=host`
  - [other options]*`$ROOT/SRC/llvm-project-llvmorg-17.0.6/llvm/`

**NOTA:** L'opzione migliore per finalità di sviluppo sarebbe quella di configurare con l'opzione `-DCMAKE_BUILD_TYPE=Debug`, che renderebbe i tools debuggabili (con `gdb` e simili). Al momento però questo comporta l'aumento delle dimensioni del BUILD/INSTALL tree a oltre un centinaio di GB, quindi non è praticabile su macchine non adeguatamente equipaggiate.

# Compilazione

- `cd ROOT/BUILD` (dovreste già trovarvi qui)
- `make -j2` (l'opzione indica l'utilizzo di più cores per la compilazione)

**NOTA:** L'opzione `-j` comporta l'incremento dell'utilizzo della RAM. Verificare che il numero di cores indicato non comporti un sovraccarico della stessa ed eventualmente ridurlo (o disattivare completamente l'opzione)

# Installazione

- `cd ROOT/BUILD` (dovreste già trovarvi qui)
- `make install`
- Al termine del processo tutti i tools si troveranno installati in `$ROOT/INSTALL`





**UNIMORE**  
UNIVERSITÀ DEGLI STUDI DI  
MODENA E REGGIO EMILIA

Dipartimento di Scienze Fisiche,  
Informatiche e Matematiche

# Scrivere un passo LLVM

# Ricorda: I *passi* LLVM

- Abbiamo visto che il middle-end è organizzato come una sequenza di *passi*
  - *Passi* di analisi
    - Consumano la IR e raccolgono informazioni sul programma
  - *Passi* di trasformazione
    - Trasformano il programma e producono nuova IR
- Perché esiste questo isolamento?
  - Migliore leggibilità
  - Diversi passi potrebbero richiedere la stessa informazione
  - L'isolamento evita analisi ridondante
- Per analizzare il codice e trasformarlo occorre una IR espressiva che mantenga tutte le informazioni importanti da trasmettere da un *passo* all'altro

# La IR di LLVM

- La IR di LLVM ha una sintassi ed una semantica simile a quelle del linguaggio Assembly a cui siete abituati (es., RISC-V)

```
int main()  
{  
    return 0;  
}
```

```
define i32 @main() ...  
{  
    ret i32 0  
}
```

# Come scrivere un passo LLVM?

- Per rispondere a questa domanda bisogna prima comprendere i seguenti punti:
  - **Moduli LLVM:**
    - Come è tradotto il nostro programma in LLVM?
  - **Iteratori:**
    - Come attraversare il modulo?
  - **Downcasting:**
    - Come ricavare maggiori informazioni dagli iteratori?
  - **Interfacce dei passi LLVM:**
    - Che interfacce fornisce LLVM per scrivere i passi?





<https://llvm.org/docs/WritingAnLLVMNewPMPass.html>

<https://llvm.org/docs/WritingAnLLVMPass.html>

Legacy Pass Manager, sta gradualmente sparendo dal middle-end

# Moduli LLVM

## Il nostro programma

- Files 
- Funzioni 
- Basic Blocks 
- Istruzioni 

## Un Modulo LLVM

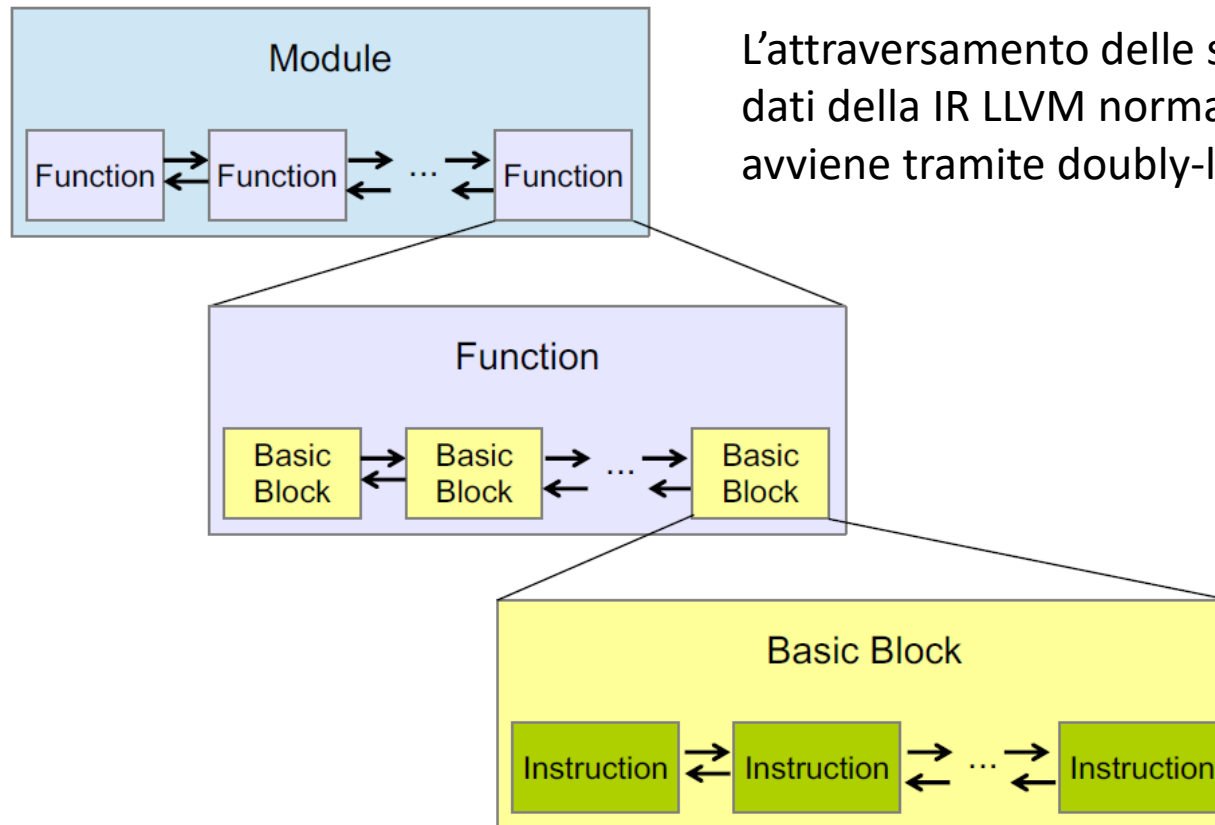
- Module
  - lista di Function e variabili globali
- Function
  - lista di BasicBlocks e argomenti
- BasicBlock
  - lista di Instruction
- Instruction
  - Opcode e operandi

# Iteratori

```
Module &M = ...;  
for (auto iter = M.begin(); iter != M.end(); ++iter) {  
    Function &F = *iter;  
    // do some stuff for each function  
}
```

- **Nota:**
  - Sintassi simile a quella del container STL `vector`.

# Iteratori



L'attraversamento delle strutture dati della IR LLVM normalmente avviene tramite doubly-linked lists

# Downcasting

- Perché ci serve il **Downcasting**?
- Immaginiamo di avere una `Instruction`
  - Come facciamo a sapere se è un'istruzione unaria o binaria?
  - O capire se è una branch instruction o una call instruction?
- Il **Downcasting** ci aiuta a recuperare maggiore informazione dagli `Iterators`
- Es.:

```
Instruction *inst = ...  
if (CallInst *call_inst = dyn_cast<CallInst>(inst))  
    outs() << "I am a call instruction"  
        << std::endl;
```

```
#include "llvm/IR/Instructions.h"
```



# Interfacce dei passi LLVM

- LLVM fornisce diverse interfacce per i passi
  - `BasicBlockPass`: itera sui *basic blocks*
  - `CallGraphSCCPass`: itera sui nodi del *call graph*
  - `FunctionPass`: itera sulla lista delle funzioni nel modulo
  - `LoopPass`: itera sui *loops*, in ordine inverso di nesting
  - `ModulePass`: generico passo interprocedurale
  - `RegionPass`: itera sulle SESE *regions*, in ordine inverso di nesting
- Come usare queste interfacce?
  - Lo vedremo più avanti

# (New) Pass manager

- Il pass manager del middle-end ha una sequenza di default di applicazione dei passi
- Il default si può alterare invocando una sequenza arbitraria di passi tramite linea di comando

```
$ opt -passes='pass1,pass2' /tmp/a.ll -S  
# -p is an alias for -passes  
$ opt -p pass1,pass2 /tmp/a.ll -S
```

<https://llvm.org/docs/NewPassManager.html>

<https://llvm.org/docs/CommandGuide/opt.html>



**UNIMORE**  
UNIVERSITÀ DEGLI STUDI DI  
MODENA E REGGIO EMILIA

Dipartimento di Scienze Fisiche,  
Informatiche e Matematiche

# **LAB 1 – TEST PASS**

# Test Pass – Sorgente

- Scaricate dalla pagina Moodle i files di test per l'esercitazione 1
  - `Fibonacci.c`
  - `Loop.c`

# Esercizio 1 – IR e CFG

- Per produrre la IR da dare in pasto al middle-end ci serve **clang** (il *frontend*)

• `clang -O2 -emit-llvm -S -c test/Loop.c \`  
`-o test/Loop.ll`

produce la IR

solo compilazione (genera assembly)

output file

input file

# Esercizio 1 – IR e CFG

- Oppure si può prima produrre il *bytecode* e poi disassemblare per produrre la forma assembly
- `clang -O2 -emit-llvm -c test/Loop.c \`  
`-o test/Loop.bc`
- `llvm-dis test/Loop.bc -o=./test/Loop.ll`

# Esercizio 1 – IR e CFG

- Per ognuno dei test benchmarks produrre la IR con clang e analizzarla, cercando di capire cosa significa ogni parte
- Disegnare il CFG per ogni funzione

# Esercizio 1 – IR e CFG

```
int g;  
  
int g_incr(int c) {  
    g += c;  
    return g;  
}
```

**SORGENTE**

```
@g = dso_local local_unnamed_addr global i32 0, align 4  
  
define dso_local i32 @g_incr(i32 @noundef %0) local_unnamed_addr #0  
{  
    %2 = load i32, i32* @g, align 4, !tbaa !3  
    %3 = add nsw i32 %2, %0  
    store i32 %3, i32* @g, align 4, !tbaa !3  
    ret i32 %3  
}
```

**LLVM IR**



# Esercizio 1 – IR e CFG

```
int g;  
  
int g_incr(int c) {  
    g += c;  
    return g;  
}
```

symbol will resolve within  
the same local unit

SORGENTE

The address of the object is not  
known within the module

```
@g = dso_local local_unnamed_addr global i32 0, align 4  
  
define dso_local i32 @g_incr(i32 noundef %0) local_unnamed_addr #0  
{  
    %2 = load i32, i32* @g, align 4, !tbaa !3  
    %3 = add nsw i32 %2, %0  
    store i32 %3, i32* @g, align 4, !tbaa !3  
    ret i32 %3  
}
```

Function definition

Attributes are described later in the file

LLVM IR

# Esercizio 1 – IR e CFG

```
; Function Attrs: nofree norecurse nosync nounwind uwtable
define dso_local i32 @loop(i32 noundef %0, i32 noundef %1, i32
noundef %2) local_unnamed_addr #1 {
    %4 = load i32, i32* @g, align 4, !tbaa !3
    %5 = icmp sgt i32 %1, %0
    br i1 %5, label %6, label %10

6:                                ; preds = %3
    %7 = sub i32 %1, %0
    %8 = mul i32 %7, %2
    %9 = add i32 %4, %8
    store i32 %9, i32* @g, align 4, !tbaa !3
    br label %10

10:                               ; preds = %6, %3
    %11 = phi i32 [ %9, %6 ], [ %4, %3 ]
    ret i32 %11 h
}
```

LLVM IR

```
int loop(int a, int b, int c) {
    int i, ret = 0;
    for (i = a; i < b; i++)
        g_incr(c);
}
```

SORGENTE

# Esercizio 1 – IR e CFG

```
; Function Attrs: nofree norecurse nosync nounwind uwtable
define dso_local i32 @loop(i32 noundef %0, i32 noundef %1, i32
noundef %2) local unnamed addr #1 {
```

```
    %4 = load i32, i32* @g, align 4, !tbaa !3
    %5 = icmp sgt i32 %1, %0
    br i1 %5, label %6, label %10
```

**BB1**

LLVM IR

```
6:                                ; preds = %3
    %7 = sub i32 %1, %0
    %8 = mul i32 %7, %2
    %9 = add i32 %4, %8
    store i32 %9, i32* @g, align 4, !tbaa !3
    br label %10
```

**BB2**

```
10:                               ; preds = %6, %3
    %11 = phi i32 [ %9, %6 ], [ %4, %3 ]
    ret i32 %11
}
```

**BB3**

```
int loop(int a, int b, int c) {
    int i, ret = 0;
    for (i = a; i < b; i++)
        g_incr(c);
}
```

**SORGENTE**

# Esercizio 1 – IR e CFG

```
; Function Attrs: nofree norecurse nosync nounwind uwtable
define dso_local i32 @loop(i32 noundef %0, i32 noundef %1, i32
noundef %2) local_unnamed_addr #1 {
    %4 = load i32, i32* @g, align 4, !tbaa !3
    %5 = icmp sgt i32 %1, %0
    br i1 %5, label %6, label %10

6:                                ; preds = %3
    %7 = sub i32 %1, %0
    %8 = mul i32 %7, %2
    %9 = add i32 %4, %8
    store i32 %9, i32* @g, align 4, !tbaa !3
    br label %10

10:                               ; preds = %6, %3
    %11 = phi i32 [ %9, %6 ], [ %4, %3 ]
    ret i32 %11
}
```

LLVM IR

**Cos'è successo al loop?**

```
int loop(int a, int b, int c) {
    int i, ret = 0;
    for (i = a; i < b; i++)
        g_incr(c);
}
```

**SORGENTE**

# Esercizio 1 – IR e CFG

- Provare a rigenerare la IR con l'opzione `-O0` al comando di invocazione di *clang*
- Cosa cambia nell'intermedio?
  - Provate a visualizzare le due IR affiancate nell'editor
- Perché?
  - Provate ad aggiungere il flag `-Rpass=. *` al comando di invocazione di *clang* (con `-O2`)

# Esercizio 2 - TestPass

<https://llvm.org/docs/WritingAnLLVMNewPMPass.html>

- Predisponiamo lo scheletro per un nuovo passo
- Tutti i passi LLVM ereditano da CRTP<sup>1</sup> mix-in `PassInfoMixin<PassT>`.
- Ogni passo deve implementare un metodo `run()` che ritorna un oggetto `PreservedAnalysis` e prende in ingresso una qualche unità IR e un analysis manager.
  - Es., per un Function pass
  - `PreservedAnalyses run(Function &F, FunctionAnalysisManager &AM);`

[1] Curiously Recurring Template Pattern [https://en.wikipedia.org/wiki/Curiously\\_recurring\\_template\\_pattern](https://en.wikipedia.org/wiki/Curiously_recurring_template_pattern)

# Esercizio 2 - TestPass

<https://llvm.org/docs/WritingAnLLVMNewPMPass.html>

- Creiamo il nostro nuovo passo come parte delle Transforms/Utils
- Prima di tutto ci serve un header file
- `cd ROOT/SRC/llvm/include/llvm/Transforms/Utils/`
- `vi TestPass.h`

```
#ifndef LLVM_TRANSFORMS_TESTPASS_H
#define LLVM_TRANSFORMS_TESTPASS_H

#include "llvm/IR/PassManager.h"

namespace llvm {


class TestPass : public PassInfoMixin<TestPass> {
public:
    PreservedAnalyses run(Function &F, FunctionAnalysisManager &AM);
};

} // namespace llvm

#endif // LLVM_TRANSFORMS_TESTPASS_H
```

# Esercizio 2 - TestPass

<https://llvm.org/docs/WritingAnLLVMNewPMPass.html>

- Creiamo il nostro nuovo passo come parte delle Transforms/Utils
- Prima di tutto ci serve un header file
- `cd ROOT/SRC/llvm/include/llvm/Transforms/Utils/`
- `vi TestPass.h`  O qualunque sia il vostro editor preferito

```
#ifndef LLVM_TRANSFORMS_TESTPASS_H
#define LLVM_TRANSFORMS_TESTPASS_H

#include "llvm/IR/PassManager.h"

namespace llvm {

class TestPass : public PassInfoMixin<TestPass> {
public:
    PreservedAnalyses run(Function &F, FunctionAnalysisManager &AM);
};

} // namespace llvm

#endif // LLVM_TRANSFORMS_TESTPASS_H
```



# Esercizio 2 - TestPass

<https://llvm.org/docs/WritingAnLLVMNewPMPass.html>

- Quindi ci serve il file sorgente principale del passo
- `cd ROOT/SRC/llvm/Transforms/Utils/`
- `vi TestPass.cpp`

```
#include "llvm/Transforms/Utils/TestPass.h"

using namespace llvm;

PreservedAnalyses TestPass::run(Function &F,
                                FunctionAnalysisManager &AM) {
    errs() << Questa funzione si chiama << F.getName() << "\n";
    return PreservedAnalyses::all();
}
```

# Esercizio 2 - TestPass

<https://llvm.org/docs/WritingAnLLVMNewPMPass.html>

- Quindi ci serve il file sorgente principale del passo
- `cd ROOT/SRC/llvm/Transforms/Utils/`
- `vi TestPass.cpp`

```
#include "llvm/Transforms/Utils/TestPass.h"

using namespace llvm;

PreservedAnalyses TestPass::run(Function &F,
                                FunctionAnalysisManager &AM) {
    errs() << Questa funzione si chiama << F.getName() << "\n";
    return PreservedAnalyses::all();
}
```

Handle al nodo IR di tipo  
Function

E non corrompe nessuna delle  
analisi già fatte sulla IR (ovvero, le  
preserva tutte)

Il nostro passo si limita a stampare  
il nome di tutte le funzioni che  
trova nel file da compilare

# Esercizio 2 - TestPass

<https://llvm.org/docs/WritingAnLLVMNewPMPass.html>

- E dobbiamo informare **Cmake** dell'esistenza del nostro passo
- `cd ROOT/SRC/llvm/Transforms/Utils/`
- `vi CMakeLists.cpp`
- Aggiungere il nome del nostro file `TestPass.cpp` seguendo l'ordine alfabetico di tutti i passi.

# Esercizio 2 - TestPass

<https://llvm.org/docs/WritingAnLLVMNewPMPass.html>

- Dobbiamo quindi "registrare" il passo nel pass manager.
- Dal momento che il pass manager è generato automaticamente, dobbiamo modificare un paio di files che controllano la generazione:
  1. Aggiungiamo questa istruzione al file  
ROOT/SRC/llvm/lib/Passes/PassRegistry.def
    - `FUNCTION_PASS("testpass", TestPass())`
    - Che registrerà il passo col nome "testpass" (useremo questa stringa per attivare il passo con `opt`)

**NOTA:** Occorre inserire l'istruzione nella sezione `FUNCTION_PASS`

Come si può vedere qui sono specificate diverse interfacce per i passi:

<code>MODULE_ANALYSIS;</code>	<code>MODULE_PASS;</code>	<code>MODULE_PASS_WITH_PARAMS;</code>
<code>CGSCC_ANALYSIS;</code>	<code>CGSCC_PASS;</code>	<code>GCSCC_PASS_WITH_PARAMS;</code>
<code>FUNCTION_ANALYSIS;</code>	<code>FUNCTION_PASS;</code>	<code>FUNCTION_PASS_WITH_PARAMS;</code>
<code>LOOP_ANALYSIS;</code>	<code>LOOP_PASS;</code>	<code>LOOP_PASS_WITH_PARAMS</code>

# Esercizio 2 - TestPass

<https://llvm.org/docs/WritingAnLLVMNewPMPass.html>

- Dobbiamo quindi "registrare" il passo nel pass manager.
- Dal momento che il pass manager è generato automaticamente, dobbiamo modificare un paio di files che controllano la generazione:
  1. Aggiungiamo questa istruzione al file  
`ROOT/SRC/llvm/lib/Passes/PassRegistry.def`
    - `FUNCTION_PASS("testpass", TestPass())`
    - Che registrerà il passo col nome "testpass" (useremo questa stringa per attivare il passo con `opt`)
  2. Aggiungiamo questa istruzione al file  
`ROOT/SRC/llvm/lib/Passes/PassBuilder.cpp`
    - `#include "llvm/Transforms/Utils/TestPass.h"`

# Esercizio 2 - TestPass

<https://llvm.org/docs/WritingAnLLVMNewPMPass.html>

- Dobbiamo quindi "registrare" il passo nel pass manager.
- Dal momento che il pass manager è generato automaticamente, dobbiamo modificare un paio di files che controllano la generazione:
  1. Aggiungiamo questa istruzione al file  
`ROOT/SRC/llvm/lib/Passes/PassRegistry.def`
    - `FUNCTION_PASS("testpass", TestPass())`
    - Che registrerà il passo col nome "testpass" (useremo questa stringa per attivare il passo con `opt`)
  2. Aggiungiamo questa istruzione al file  
`ROOT/SRC/llvm/lib/Passes/PassBuilder.cpp`
    - `#include "llvm/Transforms/Utils/TestPass.h"`
  3. Possiamo a questo punto compilare il nostro passo
    - `cd ROOT/BUILD`
    - `make opt`

**NOTA:** A questo stadio abbiamo già prodotto un eseguibile di `opt`, che si trova in  
`ROOT/BUILD/bin`

È possibile utilizzare direttamente questa versione del tool (l'installazione è opzionale)

# Esercizio 2 - TestPass

<https://llvm.org/docs/WritingAnLLVMNewPMPass.html>

- Dobbiamo quindi "registrare" il passo nel pass manager.
- Dal momento che il pass manager è generato automaticamente, dobbiamo modificare un paio di files che controllano la generazione:
  1. Aggiungiamo questa istruzione al file  
`ROOT/SRC/llvm/lib/Passes/PassRegistry.def`
    - `FUNCTION_PASS("testpass", TestPass())`
    - Che registrerà il passo col nome "testpass" (useremo questa stringa per attivare il passo con `opt`)
  2. Aggiungiamo questa istruzione al file  
`ROOT/SRC/llvm/lib/Passes/PassBuilder.cpp`
    - `#include "llvm/Transforms/Utils/TestPass.h"`
  3. Possiamo a questo punto compilare il nostro passo
    - `cd ROOT/BUILD`
    - `make opt`
  4. E installarlo (opzionale)
    - `make install`

# Esercizio 2 - TestPass

- Possiamo ora invocare l'ottimizzatore (opt) con il flag per fare overriding del pass manager di default
  - indichiamo noi la lista di passi da applicare

oppure `-p testpass`

- `opt -passes=testpass test/Loop.bc \`  
`-o test/LoopTestPass.bc`


oppure `'ll'` a seconda del formato col quale si sta lavorando



# Esercizio 2 - TestPass

- Possiamo ora invocare l'ottimizzatore (opt) con il flag per fare overriding del pass manager di default
  - indichiamo noi la lista di passi da applicare

- `opt -passes=testpass test/Loop.bc \`  
`-o test/LoopTestPass.bc`



dal momento che il nostro è un passo di analisi che non produce output (non altera la IR) possiamo disabilitare l'output

Rimpiazzare il flag `-o ...` con `-disable-output`

# Esercizio 2 - TestPass

- Se avete fatto tutto correttamente l'esecuzione del passo su `Loop.ll` dovrebbe produrre questo output:

```
g_incr  
loop
```

- Qui comincia il nostro lavoro...

# Esercizio 2 - TestPass

- Estendete il passo `TestPass` di modo che analizzi la IR e stampi alcune informazioni utili per ciascuna delle funzioni che compaiono nel programma di test
  1. Nome
  2. Numero di argomenti ('N+\*' in caso di funzione variadica)(\*)
  3. Numero di chiamate a funzione nello stesso modulo
  4. Numero di *Basic Blocks*
  5. Numero di *Istruzioni*
- (\*) es., per la funzione

```
int printf (const char *format, ...);
```

bisogna stampare '1+\*'

# Esercizio 2 - TestPass

- La documentazione è nostra amica
  - Indice delle classi per la IR LLVM
  - <https://llvm.org/doxygen/classes.html>

# Esercizio 3 - TestPass

- Trasformare il Function Pass in un Module Pass, preservando la sua funzionalità