

---

---

Uno studio sull'efficacia di checker automatici per la  
modernizzazione di codice C++

---

---

A study on the effectiveness of automatic  
checkers for modernizing C++ code



**UNIVERSITÀ DEGLI STUDI DI PARMA**

Anno Accademico 2015–2016

*Candidato:*  
Idriss Riouak

*Relatore:*  
Prof. Enea Zaffanella

*Università degli Studi di Parma  
Dipartimento di Matematica e Informatica  
Corso di laurea in Informatica*

*A mia madre e a mio padre.  
Senza loro sarebbe stato impossibile.*

*There is a pleasure in creating well-written, understandable programs.*

*There is a satisfaction in finding a program structure that tames the  
complexity of an application.*

*We enjoy seeing our algorithms expressed clearly and persuasively.*

*We also profit from our clearly written programs, for they are much more  
likely to be correct and maintainable than obscure ones.*

**- Harbison 1992**

# Indice

<b>1</b>	<b>Introduzione</b>	<b>1</b>
1.1	Obiettivo	1
1.2	Struttura di un compilatore	2
1.2.1	Analisi lessicale	3
1.2.2	Analisi sintattica	4
1.2.3	Analisi semantica	4
1.2.4	Generatore di codice intermedio	4
1.2.5	Ottimizzatore del codice	4
1.2.6	Generatore di codice	4
1.3	LLVM e Clang	5
1.4	Abstract syntax tree	7
1.4.1	AST: dichiarazioni e ast-dump	7
1.4.2	AST: tipi	10
1.4.3	AST: statements	12
1.4.4	AST: espressioni	13
1.4.5	Esempio completo	14
1.5	Clang Tidy	15
1.5.1	Struttura di un check	16
1.6	Class: FixItHint	18
1.6.1	Clang::SourceLocation	18
1.6.2	Applicare un Fix	18
<b>2</b>	<b>I check per la modernizzazione del codice</b>	<b>21</b>
2.1	modernize-avoid-bind	21
2.2	modernize-deprecated-headers	22
2.3	modernize-loop-convert	23
2.4	modernize-make-shared	24
2.5	modernize-pass-by-value	25
2.5.1	Move Semantics	25
2.5.2	Utilizzo del check	28
2.6	modernize-raw-string-literal	30
2.7	modernize-redundant-void-arg	30
2.8	modernize-replace-auto-ptr	30
2.9	modernize-shrink-to-fit	31
2.10	modernize-use-auto	32
2.11	modernize-use-bool-literals	33
2.12	modernize-use-default	34
2.13	modernize-use-emplace	34
2.14	modernize-use-nullptr	35
2.15	modernize-use-override	35
2.16	modernize-use-using	36

<b>3</b>	<b>Estendibilità dell'infrastruttura per i check</b>	<b>37</b>
3.1	Correzione di un errore in un check esistente . . . . .	37
3.1.1	Analisi dell'errore . . . . .	37
3.1.2	Analisi del codice sorgente . . . . .	38
3.1.3	Risoluzione . . . . .	39
3.1.4	Testing . . . . .	40
3.1.5	Patch . . . . .	42
3.2	Estensione di un check: Modernize Use Default . . . . .	45
3.2.1	Implementazione . . . . .	46
3.3	Creazione di un check: Modernize Use Delete . . . . .	47
3.3.1	Implementazione . . . . .	48
3.4	Modernize Use Enum Class . . . . .	51
3.4.1	Implementazione . . . . .	52
<b>4</b>	<b>Testing su PPL</b>	<b>54</b>
4.1	Parma Polyhedra Library . . . . .	54
4.2	Preparazione dell'ambiente di testing . . . . .	55
4.3	Statistiche . . . . .	58
<b>5</b>	<b>Conclusioni</b>	<b>61</b>
5.1	Sviluppi futuri . . . . .	61

## 1 Introduzione

La continua evoluzione dei linguaggi di programmazione ha reso difficile mantenere i software aggiornati alle più recenti prescrizioni e convenzioni stabilite dagli standard. Come ogni processo aziendale, la *modernizzazione del software*<sup>1</sup> rappresenta un costo. Nasce così l'esigenza d'avere strumenti che automatizzino tale processo.

*Clang Tidy* è uno software scritto in *C++*. Il suo scopo è quello di mettere a disposizione degli sviluppatori un framework flessibile per diagnosticare e correggere i più comuni errori di programmazione e l'utilizzo obsoleto dei costrutti del linguaggio.

### 1.1 Obiettivo

L'obiettivo del seguente lavoro è quello di:

1. Studiare la struttura dell'*albero di sintassi astratta*(AST) di Clang, i suoi attributi e metodi.
2. Studiare i meccanismi per interrogare l'AST di Clang.
3. Studiare i meccanismi diagnostici di Clang e di modifica del codice sorgente.
4. Studiare la struttura e l'efficacia dei check del modulo *modernize* di *Clang Tidy*.
5. Studiare l'efficacia del modulo *modernize* sulla *Parma Polyhedra Library*.

---

<sup>1</sup>Anche nota come **Legacy Modernization** [1].

## 1.2 Struttura di un compilatore

I primi calcolatori elettronici risalgono agli anni '40 ed erano programmati direttamente in linguaggio macchina per mezzo di sequenze di 0 e 1 che indicavano esplicitamente al calcolatore quali operazioni eseguire e in quale ordine.

Con il passare degli anni, è sorta la necessità di creare nuovi linguaggi di programmazione più ad alto livello e comprensibili all'uomo. Con lo sviluppo di quest'ultimi, è nata l'esigenza di creare un programma che preso in input un codice sorgente scritto in un linguaggio di programmazione, restituisse lo stesso, tradotto in un'altro linguaggio. Tale programma è detto *compilatore*.

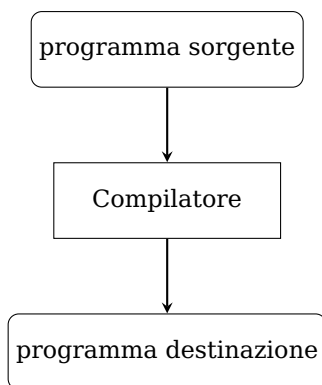


Figura 1: Macro schema di un compilatore.

Di un compilatore possiamo individuare due ulteriori macro aree: *analisi* e *sintesi*. La parte di *analisi* ha lo scopo di suddividere il programma sorgente in elementi di base e di imporre loro una struttura grammaticale, individuando dunque costrutti sintatticamente scorretti o semanticamente privi di significato. La parte di *analisi* raccoglie inoltre informazioni sul programma sorgente e le memorizza in una struttura dati detta *tabella dei simboli*. La macro area di *sintesi*, a partire dalle informazioni presenti nella tabella dei simboli e dalla rappresentazione intermedia, costruisce il programma di destinazione [2].

La parte di *analisi* è detta *front-end* mentre la parte di *sintesi* è detta *back-end*.

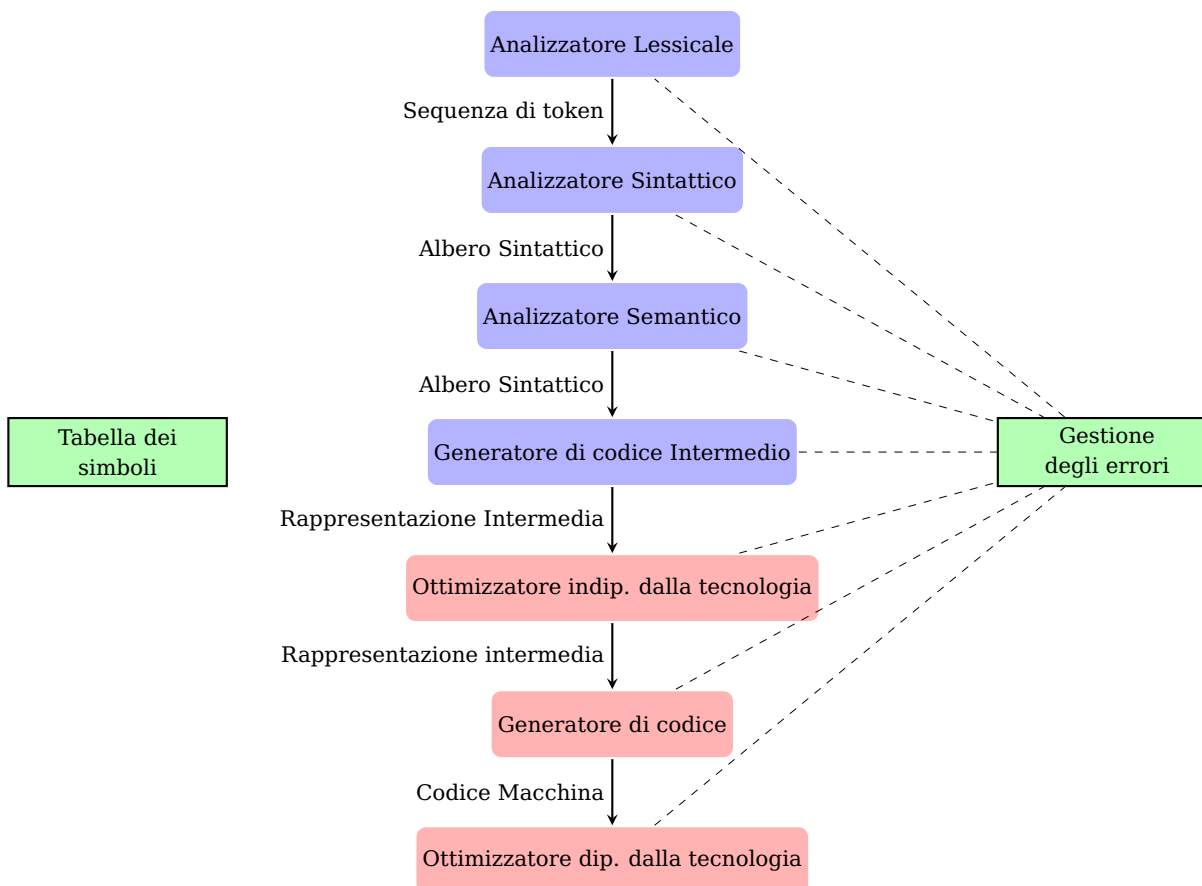


Figura 2: Fasi di compilazione: In blu il **front-end** ed in rosso il **back-end**

### 1.2.1 Analisi lessicale

Prima di poter dare una definizione di *Analizzatore lessicale*, dobbiamo definire cosa si intende per *token* e *lessema*.

Un *token* è una coppia avente la seguente forma:

< Nome-del-Token, Valore-del-Token >

Il *Nome-del-Token* è un simbolo astratto che rappresenta un'unità lessicale quale un identificatore oppure una *Keyword* riservata dal linguaggio. Un *lessema* è una sequenza di caratteri presente nel codice sorgente, corrispondente ad un'istanza di un token [3].

La prima fase di compilazione è detta *analisi lessicale*. Questa fase prende in input il codice sorgente del programma da compilare, come uno stream di caratteri, scarta tutti gli spazi bianchi in eccesso ed i commenti, e



per ogni lessema, l'analizzatore produce un token. Il risultato viene passato all'*analizzatore sintattico*.

### 1.2.2 Analisi sintattica

L'*analizzatore sintattico*, utilizza il primo elemento dei token prodotti dall'*analizzatore lessicale* per produrre una rappresentazione intermedia ad albero che indica la struttura grammaticale della sequenza di token. Il risultato di tale fase è l'*albero sintattico*.<sup>2</sup>

### 1.2.3 Analisi semantica

L'*analizzatore semantico* utilizza l'albero sintattico per verificare la consistenza del programma sorgente rispetto alla definizione del linguaggio: una parte importante dell'analisi semantica è infatti il *controllo sui tipi*, ovvero verificare che ogni operatore abbia operandi di tipo compatibile tra loro.

### 1.2.4 Generatore di codice intermedio

Nel processo di traduzione di un programma sorgente in un programma destinazione, il compilatore costruisce una o più rappresentazioni intermedie che possono assumere diverse forme.<sup>3</sup>

Dopo tali fasi molti compilatori generano esplicitamente una rappresentazione di basso livello, simile al codice macchina.

### 1.2.5 Ottimizzatore del codice

La fase di ottimizzazione del codice, indipendente dalla macchina di destinazione, cerca di migliorare il codice intermedio in termini di velocità d'esecuzione e di dimensione durante la generazione del codice destinazione.

### 1.2.6 Generatore di codice

Il generatore di codice prende in ingresso la rappresentazione intermedia, ottenuta dalla fase di ottimizzazione, e la traduce nel programma destinazione.

---

<sup>2</sup>Noto anche come Syntax Tree.

<sup>3</sup> L'albero sintattico è una di queste.

### 1.3 LLVM e Clang

La *Low-Level Virtual Machine (LLVM)* è l'infrastruttura completa di un compilatore, progettato da zero per consentire un'ottimizzazione a più fasi. Il progetto è stato avviato nel 2000 da Chris Lattner, docente dell'Università dell'Illinois. LLVM ha attirato sempre di più l'attenzione, specialmente quando, nel 2005 *Apple Inc.* ha preso le redini del progetto assumendo Chris Lattner.

LLVM è completamente scritta in *C++*, ed è rilasciata sotto licenza BSD, che lo rende dunque un software *open source*. Essendo LLVM solamente un'infrastruttura, non è utilizzabile come compilatore, infatti necessita di un *front end*. Il front-end più utilizzato per i linguaggi <sup>4</sup> *C* e i suoi dialetti è *Clang*.

Sebbene inizialmente il nome Clang si riferisse solo al front-end, con il tempo è passato a denominare il compilatore completo, ottenuto combinando il front-end con il back-end fornito da LLVM.

Clang è un compilatore open source per i linguaggi *C*, *C++*, *Objective C*, *Objective C++* e tutti i loro dialetti. E' stato costruito su LLVM con l'obiettivo di offrire un maggiore supporto ai linguaggi *Objective C/C++*, che venivano presi poco in considerazione dagli sviluppatori di GCC.

Oltre ad essere un compilatore, Clang mette a disposizione uno strumento d'analisi statica [4]., ovvero una suite di tecniche e algoritmi usate per analizzare il codice sorgente dell'utente, con il fine di trovare e segnalare a tempo di compilazione gli errori che si potrebbero presentare a tempo d'esecuzione.

Rispetto al front end *LLVM-GCC*, *Clang* ha le seguenti proprietà:

- **Diagnostica dettagliata:** Gli *errori* e i *warnings* sono più descrittivi e dettagliati.
- **2.5x Faster:** E' stato testato<sup>5</sup>, che tutte le fasi di Clang, nel complesso, sono 2.5 volte più veloci [5] rispetto a quelle di LLVM-GCC.
- **Modularità:** Clang, come LLVM, gode di una forte modularità, dunque riusabilità del codice e facile aggiunta di estensioni.

---

<sup>4</sup>Ma non unico, in quanto esistono altri *front end* utilizzati per analizzare linguaggi differenti come ad esempio **LLVM-GCC** per il linguaggio Fortran.

<sup>5</sup>Dati presi dalla presentazione di Clang alla conferenza **LLVM & Apple: New LLVM C front end**.

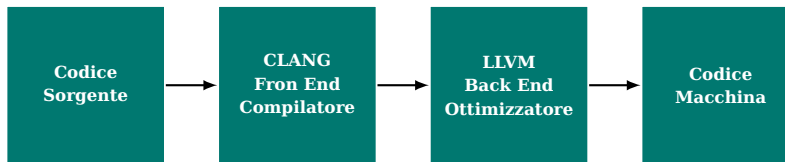


Figura 3: Clang e LLVM.

Lo sviluppo di Clang, è stato un importante stimolo per migliorare GCC. Al giorno d'oggi, infatti, la diagnostica di GCC è decisamente migliorata e sono stati fatti netti miglioramenti anche sulla velocità del parser.

## 1.4 Abstract syntax tree

L'albero di sintassi astratta, d'ora in avanti AST, è il cuore pulsante di Clang, in quanto la rappresentazione finale generata dalla parte di *analisi* del *front-end* è un albero di sintassi. L'AST è una rappresentazione più ad alto livello del programma utilizzata dal compilatore. In particolare viene usato per riempire la tabella dei simboli e per semplificare il controllo dei tipi (*type check*).

L'AST di Clang è vasto, in quanto deve rappresentare tutti i possibili tipi (*Type*), espressioni (*Expr*), dichiarazioni (*Decl*) e statement (*Stmt*) del C++, infatti la linea di sviluppo tende all'ottimizzazione del codice interno, per diminuire lo spazio in memoria occupato.

Il punto d'ingresso per entrare nell'albero di sintassi astratta è la funzione **getTranslationUnitDecl()** che restituisce l'unità di traduzione sulla quale si sta agendo, permettendo di lavorare sulle principali classi dell'AST.

### 1.4.1 AST: dichiarazioni e ast-dump

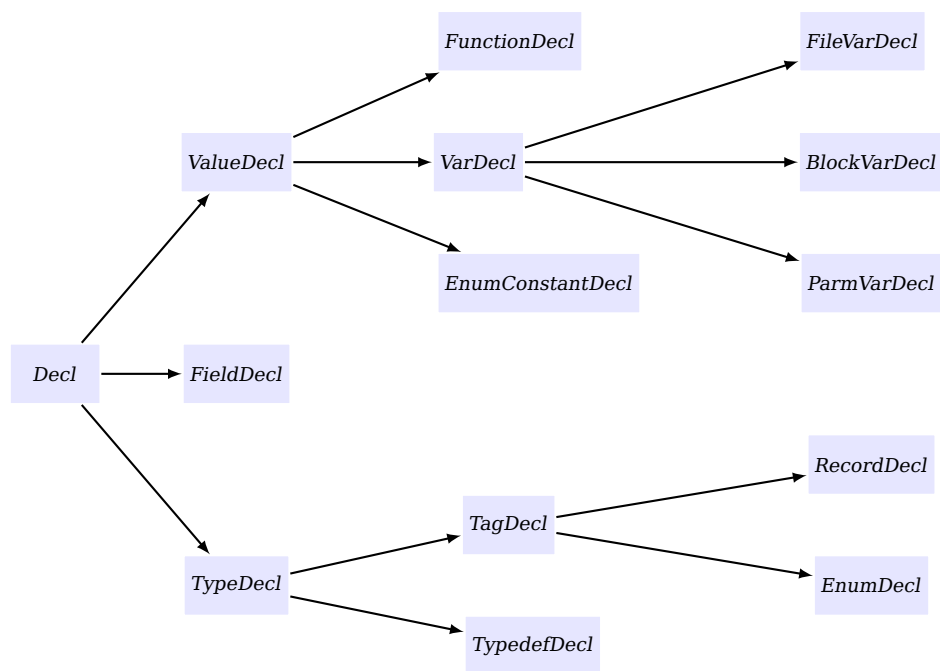


Figura 4: Struttura della macro classe Decl, con le principali classi derivate, dell'albero di sintassi astratta di Clang.

Come si può vedere dallo schema in *figura 4*, qualunque tipo di dichiarazione deriva dalla classe *Decl*.

In particolare possiamo distinguere tre macro categorie:

- **ValueDecl:** rappresenta la dichiarazione di una variabile, funzione o di un'enumerazione costante.
- **FieldDecl:** un'istanza di tale classe rappresenta un campo di una *struct/union/class*.
- **TypeDecl:** rappresenta la dichiarazione di un tipo. Quest'ultima può avvenire sia con il costrutto *typedef* che con la dichiarazione di una *struct/union/class*. *Clang* permette di visualizzare l'AST di un'unità di traduzione direttamente da terminale, mostrando come i lessemi del codice sorgente vengono rappresentati attraverso le classi dell'AST.

Consideriamo la seguente porzione di codice: **[Esempio #1.1]**

```
//Dichiarazione di una variabile
int x;
//Dichiarazione di un'enumerazione costante
enum Colori{
    Rosso
};
//Dichiarazione di una funzione
void foo();
//Dichiarazione di una classe
class Bar{};
//Dichiarazione di un nuovo tipo
typedef short BYTE;
```

Lanciando il seguente comando da terminale:

```
clang -std=C++11 -Xclang -ast-dump main.cpp
```

Otteniamo l'albero di sintassi astratta generato da *Clang*:

```
TranslationUnitDecl 0x7fd85081ded0 <<invalid sloc>> <invalid sloc>
|-TypeDecl 0x7fd85081e408 <<invalid sloc>> <invalid sloc> implicit __int128_t '__int128'
|`-BuiltinType 0x7fd85081e120 '__int128'
|-TypeDecl 0x7fd85081e468 <<invalid sloc>> <invalid sloc> implicit uint128_t 'unsigned __int128'
|`-BuiltinType 0x7fd85081e140 'unsigned __int128'
|-TypeDecl 0x7fd85081e798 <<invalid sloc>> <invalid sloc> implicit NSString 'struct NSString_tag'
|`-RecordType 0x7fd85081e550 'struct NSString_tag'
|`-CXXRecord 0x7fd85081e4b8 'NSString_tag'
|-TypeDecl 0x7fd85081e828 <<invalid sloc>> <invalid sloc> implicit builtin_ms_va_list 'char *'
|`-PointerType 0x7fd85081e7f0 'char *'
|`-BuiltinType 0x7fd85081df60 'char'
|-TypeDecl 0x7fd85081eb48 <<invalid sloc>> <invalid sloc> implicit builtin_va_list 'struct __va_list_tag [1]'
|`-ConstantArrayType 0x7fd85081eaf0 'struct __va_list_tag [1]' 1
|`-RecordType 0x7fd85081e910 'struct __va_list_tag'
|`-CXXRecord 0x7fd85081e878 '__va_list_tag'
|-VarDecl 0x7fd850868e00 <main.cpp:1:1, col:5> col:5 x 'int'
|-EnumDecl 0x7fd850868ea0 <line:3:1, line:5:1> line:3:6 Colori
|`-EnumConstantDecl 0x7fd850868f60 <line:4:5> col:5 Rosso 'enum Colori'
|-FunctionDecl 0x7fd850869000 <line:7:1, col:10> col:6 foo 'void (void)''
|`-CXXRecordDecl 0x7fd8508690a8 <line:9:1, col:11> col:7 class Bar definition
|`-CXXRecordDecl 0x7fd8508691c0 <col:1, col:7> col:7 implicit class Bar
|-TypeDecl 0x7fd850869268 <line:11:1, col:15> col:15 BYTE 'short'
|`-BuiltinType 0x7fd85081dfa0 'short'
```

Figura 5: Dump dell'AST sul file *main.cpp*

Come si può vedere, per la dichiarazione della variabile *x* si ha un nodo di tipo *VarDecl*, per l'enumerazione un *EnumDecl* (classe contenente un *EnumConstantDecl*), per la funzione *foo* una *FunctionDecl*, per la classe *Bar* un *CXXRecordDecl* e per la definizione del tipo *BYTE* un nodo di tipo *TypeDecl*.

Questo strumento è utile sotto molti aspetti, in particolare aiuta il programmatore che lavora sull'AST a capire quali tipi di nodi sono presenti nel codice sorgente.

### 1.4.2 AST: tipi

In generale, il C++ si caratterizza per una certa complessità [6] dei meccanismi per costruire tipi di dato.

Si consideri il seguente esempio:

**[Esempio #1.2]**

```
int x;  
const int y = Intero;
```

La parola chiave *int* indica il tipo delle variabili dichiarate, mentre *const* è detto qualificatore. *Clang* gestisce i tipi creando un oggetto di tipo *QualType*, che contiene il qualificatore del tipo e il puntatore al tipo.

Si consideri ora un esempio più complicato da gestire: i tipi nidificati.

**[Esempio #1.3]**

```
1 int A = 1;  
2 int* const B = &A;  
3 int* const C = {B};  
4 int* const* D = &C;
```

In questi casi (linee 2, 3 e 4), il qualificatore *const* appare al centro della dichiarazione, e dunque risulta complicato conoscere il tipo della variabile alla quale fa riferimento. Per evitare di creare tutte le possibili combinazioni, il problema viene gestito utilizzando ***PointerType::GetPointeeType() const***, che restituisce un elemento di tipo *QualType*, che a sua volta può restituire il tipo della variabile.

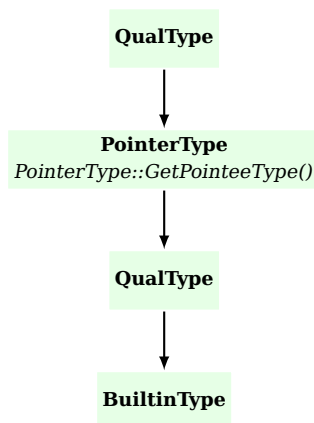


Figura 6: Gestione dei tipi annidati in *Clang*

Di seguito la struttura gestione dei tipi nell'AST.

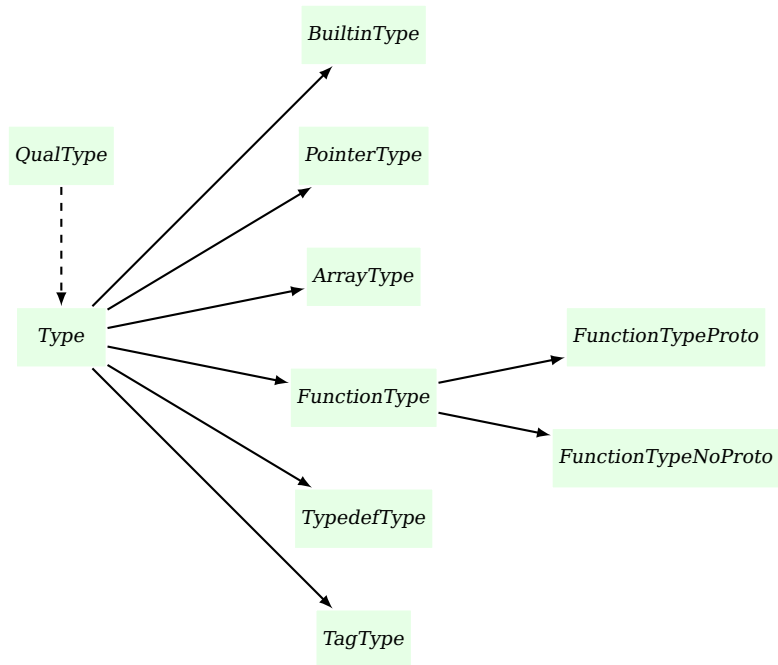


Figura 7: Struttura della macro classe `Type`, con le principali classi derivate, dell'albero di sintassi astratta di Clang.



### 1.4.3 AST: statements

La classe **Stmt** rappresenta uno statement del C++. La parte interessante di tali oggetti è la possibilità di interrogarli, come ad esempio l'*IfStmt*, dove possiamo invocare i metodi *getCond()*, *getThen()* e *getElse()*, che rappresentano rispettivamente la condizione, il ramo *then* e il ramo *else* del costruito condizionale.

Di seguito la struttura completa e divisa per tipologie di tutti gli *Stmt* rappresentabili:

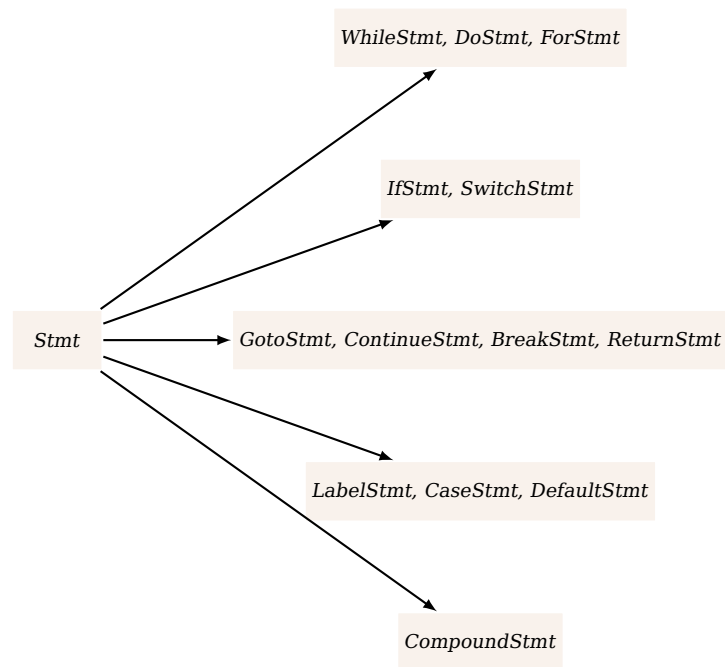


Figura 8: Struttura della macro classe Stmt, con tutte le classi derivate e divisi per tipologia, dell'albero di sintassi astratta di Clang.

Lo statement più utilizzato è il *CompoundStatement*, che rappresenta un gruppo di statement come `{ Stmt Stmt }`. Consideriamo il seguente esempio, concentrandoci solamente sulla classe **Stmt**. **[Esempio #1.4]**

```
int main() {  
    int param = 1;  
    return 0;  
}
```

Da questo frammento di codice possiamo creare il seguente albero contenente la dichiarazione della funzione *main* e tutti gli Stmt presenti:

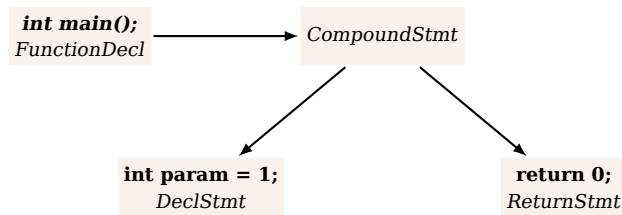


Figura 9: Esempio di CompoundStmt

#### 1.4.4 AST: espressioni

Le espressioni vengono rappresentate come una sottoclasse di *Stmt*. Questo permette di utilizzare un'espressione ovunque possa apparire uno statement.

Di seguito tutte le sottoclassi di *Expr*:

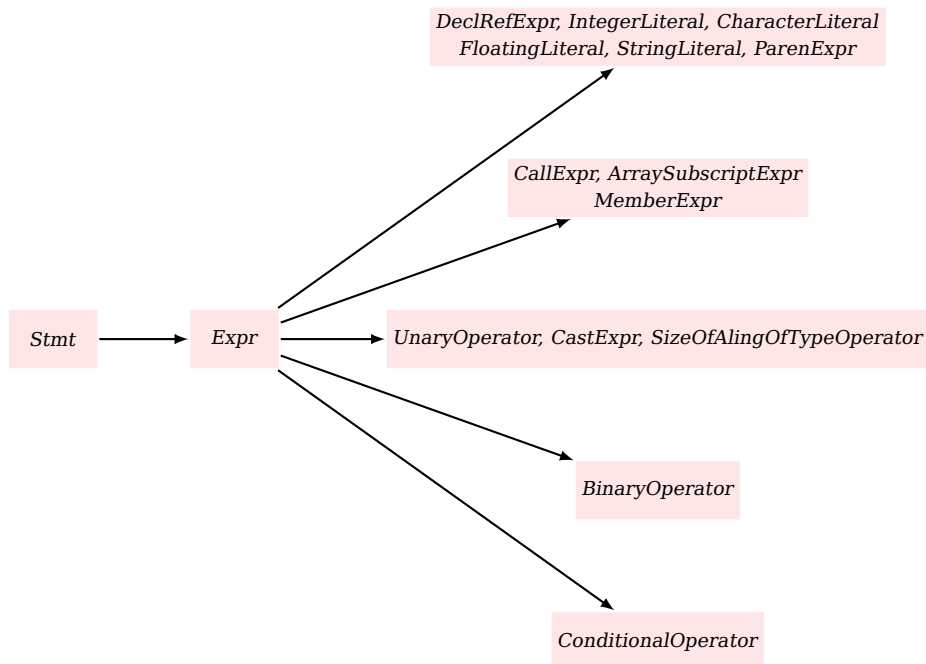


Figura 10: Struttura della macro classe Expr, con le principali classi derivate, dell'albero di sintassi astratta di Clang.

## 1.4.5 Esempio completo

**[Esempio #1.5]**

Consideriamo il frammento di codice dell'esempio #1.4, con l'aggiunta di una chiamata a funzione:

```
void SetStream(int);

int main(int argc, char *argv[]) {
    int StreamNumber = 1;
    SetStream(StreamNumber);
    return 0;
}
```

L'AST generato è il seguente:

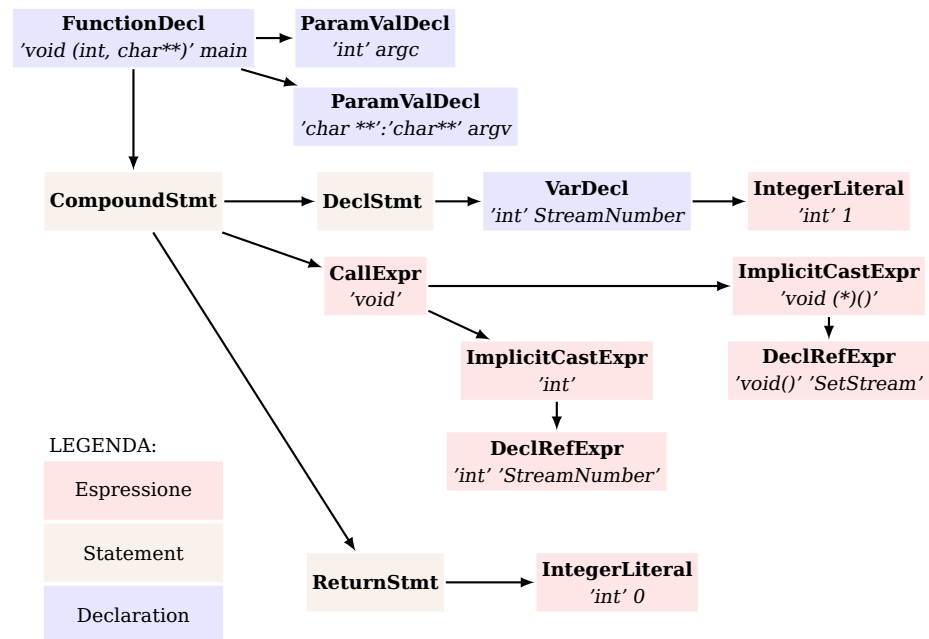


Figura 11: Albero di sintassi astratta del codice.

## 1.5 Clang Tidy

Clang venne annunciato durante la presentazione della *Apple and LLVM community* [5] del 2007. Gli obiettivi prefissati dal team di sviluppo erano:

- Creazione di un parser unificato per i linguaggi di tipo C.
- Alte prestazioni.
- Multiuso: strumenti per la generazione del codice, l'analisi statica e l'indicizzazione.

L'implementazione di quest'ultimo punto si verifica con la nascita dei *Clang-Tools*, una suite di strumenti indipendenti tra loro utilizzabili direttamente da riga di comando. Tutti gli strumenti vengono organizzati in tre macro categorie, due delle quali risiedono nella main-repository del progetto <sup>6</sup>, la restante è collocata in un'extra-repository essendo principalmente utilizzata dagli sviluppatori più esperti.

Le tre macro categorie sono le seguenti:

- **clang-check**: analizzatore statico che a tempo di compilazione, se utilizzato con l'opzione *-analyze*, effettua analisi di vario tipo (controllo di alcuni errori in fase di assegnamento alle variabili, controllo sui bound degli array, ecc.).
- **clang-format**: è sia una libreria che uno strumento indipendente da tutto il front-end con l'unico scopo di formattare automaticamente i sorgenti scritti in C++.
- **clang-tidy (Extra Clang Tools)**: strumento utilizzato per individuare e risolvere i più frequenti errori di programmazione, aiutando lo sviluppatore a seguire determinati pattern ed utilizzare nuovi costrutti descritti nello standard C++11, in sostituzione di pattern considerati inefficienti, ineleganti, obsoleti o addirittura deprecati in quanto facilmente soggetti ad errori di programmazione.

Noi focalizzeremo l'attenzione su *Clang Tidy*.

Come già detto, Clang Tidy è uno strumento progettato dagli sviluppatori di *Clang & LLVM*. Contiene una serie di check, che sono raggruppati nei seguenti moduli:

- **llvm-**: check relativi alle convenzioni sulla scrittura del codice imposte da LLVM.
- **google-**: check relativi alle convenzioni sulla scrittura del codice imposte da *Google*.

---

<sup>6</sup>Perché utilizzate principalmente dagli IDE (Xcode, Emacs, Vim, ecc.).

- **modernize-\***: check utilizzati per modernizzare<sup>7</sup> i file sorgenti, utilizzando i nuovi costrutti.
- **redability-**: check il cui scopo è quello di risolvere i problemi legati alla leggibilità del codice.
- **misc-**: tutti i check che non rientrano in una delle categorie sopra citate.
- **clang-analyzer-**: check di analisi statica di Clang.
- **boost-**: check legati alla libreria *Boost*

Data la grande quantità di check messi a disposizione da Clang Tidy, l'attenzione verrà concentrata sul gruppo **Modernize**, in particolare verrà analizzata la struttura e le caratteristiche di ogni check e il meccanismo di segnalazione e correzione degli errori.

### 1.5.1 Struttura di un check

Ogni check di Clang Tidy è strutturato nel seguente modo:

- Per ogni check presente in Clang Tidy, esiste una classe che ne descrive il comportamento ed è così strutturata:

```
#include "../ClangTidy.h"

namespace clang {
namespace tidy {
namespace some_module8 {

class MyCheck : public ClangTidyCheck {
public:
    MyCheck(StringRef Name, ClangTidyContext *Context):
        ClangTidyCheck(Name, Context) {}
    //Other Methods...
};

} // namespace some_module
} // namespace tidy
} // namespace clang
```

Come si può vedere il costruttore del check riceve il *Nome* e il *Contesto* nel quale viene invocato il check, che vengono inoltrati al costruttore della classe base ***ClangTidyCheck***.

---

<sup>7</sup>Attualmente *moderno* significa utilizzare i costrutti dello standard **C++11** [7].

<sup>8</sup>Per **some\_module**: uno tra quelli citati nel capitolo 1.5

- **AST Matcher:** per potere lavorare sugli elementi dell'AST bisogna effettuare l'overriding di due metodi:

```
void registerMatchers(ast_matchers::MatchFinder *Finder)
    override;
```

Effettuando l'overriding di tale metodo, si può specificare quali nodi dell'AST il check è interessato ad analizzare, etichettandoli in modo tale da potervi accedervi, come nel seguente esempio:

**[Esempio #1.6.1]**

```
static const char NamespaceDeclaration[] =
    "NamespaceDeclaration";
void ReplaceNamespaceClass::registerMatchers(MatchFinder
    *Finder) {
    if (getLangOpts().Cplusplus9) {
        Finder->addMatcher(namespaceDecl().
            bind(NamespaceDeclaration), this);
    }
}
```

Il seguente esempio, etichetta con il nome *"NamespaceDeclaration"* ogni nodo dell'AST che corrisponde ad una dichiarazione di un *namespace*.

Dopo aver trovato ed etichettato i nodi dell'AST utili al check, bisogna effettuare l'overriding di un ulteriore metodo per poter utilizzare tali nodi. Il metodo è il seguente:

```
void check(const ast_matchers::MatchFinder::MatchResult
    &Result) override;
```

Infatti, la variabile *Result* contiene un campo *Node*, che con una chiamata a funzione può restituire i nodi in base al nome con la quale sono stati etichettati.

Di seguito il continuo dell'esempio **1.6.1**. **[Esempio #1.6.2]**

```
void ReplaceNamespaceClass::check(const
    MatchFinder::MatchResult &Result) {
    const NamespaceDecl *NSD = Result.Nodes
        .getNodeAs<NamespaceDecl>(NamespaceDeclaration);
    //Do Something with NSD.
}
```

---

<sup>9</sup>Per assicurarci che il linguaggio di programmazione sulla quale il check agisce sia C++

I nodi vengono restituiti in ordine identico a quello con il quale sono stati registrati nel metodo `registerMatchers()`.

## 1.6 Class: `FixItHint`

Nel contesto di *Clang-Tidy*, la classe `FixItHint` è estremamente importante in quanto permette di gestire le modifiche da effettuare sul codice sorgente. In particolare, specificando l'opzione `-fix` o `-fix-errors`<sup>10</sup>, viene attivata la gestione degli errori.

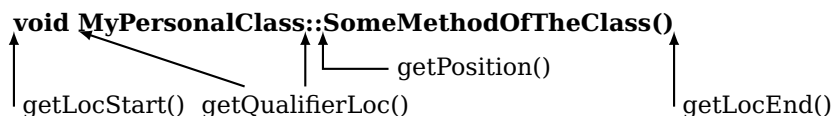
Prima di poter descrivere il comportamento di tale classe, introduciamo il concetto di `SourceLocation`.

### 1.6.1 `Clang::SourceLocation`

La classe `SourceLocation` codifica la posizione (file, riga, colonna) degli elementi dell'AST presenti nel codice sorgente.

Consideriamo il seguente esempio:

**[Esempio #1.7]**



La seguente funzione viene rappresentata nell'AST come un nodo di tipo `CXXMethodDecl` e tutti i metodi (`getLocStart()`, `getQualifierLoc()`, `getPosition()`, `getLocEnd()`) sono metodi dell'omonima classe: `CXXMethodDecl`.

Quasi tutti gli oggetti dell'AST hanno metodi che restituiscono delle "locazioni". In tal modo possiamo conoscere la posizione esatta di un oggetto dell'AST all'interno del codice sorgente.

### 1.6.2 Applicare un Fix

Ora si hanno tutti gli elementi per poter effettuare un Fix.

Per poter segnalare all'utente la presenza di un possibile Fix, si utilizza il metodo `Diag` della classe `ClangTidyCheck` nel seguente modo:

```
ClangTidyCheck::diag(SourceLocation Loc, StringRef Message,  
DiagnosticIDs::Level Level)
```

Con il seguente significato per i parametri:

---

<sup>10</sup>`-fix` e `-fix-errors` sono due opzioni completamente diverse. La prima applica i fix proposti da Clang-Tidy ad esempio utilizzando il package `modernize`, mentre `-fix-errors` prova a correggere gli errori riscontrati a tempo di compilazione. `-fix-errors` implica `-fix`.

- **Loc** rappresenta il punto (file, riga, colonna) nel codice sorgente dove il *Fix* potrebbe essere applicato.
- **Message** è una stringa contenente la descrizione dell'errore.
- **Level** rappresenta il nome della classe che ha proposto il *Fix*.

Mentre per rendere effettivo un fix e dunque modificare il codice sorgente, si utilizza la classe **FixItHint**. Questa classe mette a disposizione dei metodi statici per inserire, rimuovere o sostituire frammenti di codice.

I metodi della classe `FixItHint` sono i seguenti:

- **CreateInsertion(SourceLocation, StringRef, bool)**: modifica il codice sorgente inserendo la stringa `data` nella specifica location.
- **CreateRemoval(CharSourceRange RemoveRange)**: modifica il codice sorgente rimuovendo tutto ciò che è compreso nel source range.
- **CreateReplacement(CharSourceRange RemoveRange, StringRef Code)**: modifica il codice sorgente sostituendo tutto ciò che è presente nel source range con la stringa `data`.

Di seguito un estratto del codice sorgente preso dalla classe `UseDeleteCheck.cpp`: **[Esempio #1.8]**

```
std::string SpecialFunctionName;
const auto *SpecialFunctionDecl =
    Result.Nodes.getNodeAs<CXXMethodDecl>(SpecialFunction);
//Some Code ...
diag(SpecialFunctionDecl->getLocEnd(), "Use '= delete' to
    disable the " + SpecialFunctionName) <<
    FixItHint::CreateReplacement(CharSourceRange::getTokenRange(
        SpecialFunctionDecl->getLocEnd(),
        SpecialFunctionDecl->getLocEnd()), "= delete");
```

Che applicato sul file `Affine_Space.cc` della PPL con l'opzione `-header-filter=.`<sup>11</sup> rileva 11 possibili applicazioni. Di seguito uno dei warning generati da Clang-Tidy:

- **File, Riga e Colonna**:/PPL/src/Temp\_defs.hh:119:45
- **Tipo**: Warning<sup>12</sup>
- **Descrizione** Use '= delete' to disable the copy constructor [modernize-use-delete]

---

<sup>11</sup>Mostra e applica (se attiva l'opzione `-fix`) i *warning* anche sugli header file dell'utente (non system header).

<sup>12</sup>Il tipo può essere Warning oppure Error



- **Posizione del warning:**Temp\_Value\_Holder(const Temp\_Value\_Holder&);
- **Fix suggerito:** =delete

Effettivamente nella riga 119 del file Temp\_defs.hh riscontriamo la presenza di un costruttore privato e non implementato.

```
template <typename T>
class Temp_Value_Holder {
private:
    //! Copy constructor: private and intentionally not
    implemented.
    Temp_Value_Holder(const Temp_Value_Holder&);
}
```

Dunque una volta verificata la veridicità del warning generato da Clang Tidy, utilizzando l'opzione *-fix*, viene attuata la modifica del codice sorgente.

```
template <typename T>
class Temp_Value_Holder {
private:
    //! Copy constructor: private and intentionally not
    implemented.
    Temp_Value_Holder(const Temp_Value_Holder&)=delete;
}
```

## 2 I check per la modernizzazione del codice

Clang Tidy mette a disposizione diciassette check con prefisso **modernize-**. Di seguito, in ordine alfabetico, verranno descritti con alcuni esempi questi check.

### 2.1 modernize-avoid-bind

Il seguente check trova l'utilizzo di **std::bind** e lo sostituisce con una funzione *lambda*. **[Esempio #2.1]**

```
#include <iostream>
double Divisione(double x, double y);

int main(int argc, const char * argv[]) {
    using namespace std::placeholders;

    auto FBind1=std::bind(Divisione,10,2);
    auto FBind2=std::bind(Divisione, _1,2);
    auto FBind3=std::bind(Divisione, _1,_2);
}
```

Il codice sorgente scritto sopra crea un legame tra la funzione *Divisione* e le Funzioni *FBind*, cambiando il metodo con la quale vengono passati i parametri.

Prima di poter eseguire Clang Tidy su tale porzione di codice è necessario creare un *Compilation Database*, un semplice file json che descrive i comandi di compilazione, i file e la directory sulla quale Clang andrà ad agire. Di seguito il file *compile\_commands.json* utilizzato per compilare il codice sorgente visto sopra.

```
[{ "directory": "/modernize-avoid-bind/",
  "command": "clang++ -std=c++14 -c -o main.o main.cpp",
  "file": "main.cpp" }]
```

Lanciando da terminale il seguente comando:

```
clang-tidy main.cpp -checks="modernize-avoid-bind"
```

il tool genera tre warning, uno per ogni occorrenza di *std::bind* con la seguente descrizione:

```
warning: prefer a lambda to std::bind [modernize-avoid-bind]
```

Aggiungendo al comando precedente l'opzione `-fix`, Clang Tidy provvederà a sostituire ogni occorrenza di `std::bind` con una lambda funzione.

Di seguito il codice sorgente generato dopo aver rilanciato il comando

```
#include <iostream>
double Divisione(double x, double y);

int main(int argc, const char * argv[]) {
    using namespace std::placeholders;

    auto FBind1=[] { return Divisione(10, 2); };
    auto FBind2=[](auto && arg1) { return Divisione(arg1, 2); };
    auto FBind3=[](auto && arg1, auto && arg2) { return
        Divisione(arg1, arg2); };
}
```

Lo scopo di questo check è di eliminare alcune inefficienze potenziali dovute all'uso della funzione di libreria `std::bind`, preferendo l'uso delle espressioni lambda introdotte con lo standard C++11.

## 2.2 modernize-deprecated-headers

Con l'avvento dello standard *C++14*, per questioni di compatibilità con la **C standard Library** e con la *C Unicode TR*, la *C++ Standard Library* fornisce i seguenti 25 *C* header [8]:

<assert.h>	<iso646.h>	<stdarg.h>	<tgmath.h> <sup>Dep.</sup>
<complex.h>	<limits.h>	<stdbool.h> <sup>Dep.</sup>	<time.h>
<ctype.h>	<locale.h>	<stddef.h>	<uchar.h> <sup>Dep.</sup>
<errno.h>	<math.h>	<stdint.h>	<wchar.h>
<fenv.h> <sup>Dep.</sup>	<setjmp.h>	<stdio.h>	<wctype.h>
<float.h>	<signal.h>	<stdlib.h>	
<inttypes.h>	<stdalign.h> <sup>Dep.</sup>	<string.h>	

Per convenzione nel linguaggio *C* ogni header file ha un nome con la struttura *nomeHeader.h*, mentre in *C++* è *cnomeHeader*.

Clang Tidy mette a disposizione il check *modernize-deprecated-headers* che analizza il codice sorgente datogli in input, e sostituisce ogni occorrenza di header file di tipo *C* con le rispettive in *C++*.

### 2.3 modernize-loop-convert

Questo check converte i classici cicli `for( ... ; ... ; ... )` in quelli basati su intervallo introdotti con lo standard **C++11**. **[Esempio #2.3.1]**

```
std::vector<int> v(10,100);
for (std::vector<int>::iterator it=v.begin();it!=v.end();++it){
    std::cout<<*it<<std::endl;
}
```

Applicandoci il check **modernize-loop-convert** diventa:

```
std::vector<int> v(10,100);
for (int & it : v){
    std::cout<<it<<std::endl;
}
```

In particolare converte tre tipi di cicli:

- Loop su array allocati staticamente.
- Loop su contenitori della *STL* che utilizzano gli iteratori.
- Loop su contenitori utilizzando le istruzioni `operator[]` e `at()`.

Ogni conversione rischia di cambiare la semantica del loop sulla quale agisce. Infatti prima di effettuare la conversione, ogni loop viene etichettato con una delle seguenti diciture:

- **risky**: per tutti i cicli dove l'espressione che restituisce il contenitore è più complessa della sola dereferenziazione, oppure parte di essa appare anche nel corpo del ciclo rischiando di modificare la semantica del programma in caso di conversione. **[Esempio #2.3.2]**

```
int arr[10][20];
int l = 5;
for (int j = 0; j < 20; ++j)
    int k = arr[l][j] + l; //l'utilizzo di l al di fuori di
                          arr[l] e' da considerarsi "risky"
```

Lanciando il checker su tale porzione di codice non viene effettuata alcuna modifica al file sorgente. Per forzare l'applicazione delle modifiche è possibile settare il livello di *confidenza* a *risky*.

- **Reasonable (Default)**: se alla fine di ogni iterazione del ciclo vengono chiamate le funzioni membro di un contenitore `.end()` o `.size()`,

il loop è da considerarsi *accettabile* e questo verrà convertito in un *range-based for*.

Di seguito un esempio di ciclo *Reasonable*: **[Esempio #2.3.3]**

```
std::vector<int> v(10,100);
for (int i=0;i<v.size();++i){
    std::cout<<v[i]<<std::endl;
}
```

viene convertito in:

```
std::vector<int> v(10,100);
for (int i : v){
    std::cout << i<<std::endl;
}
```

Il risultato è una porzione di codice decisamente più leggibile e di facile comprensione.

- **safe**: questa categoria cattura tutti i casi restanti. **[Esempio #2.3.4]**

```
int TestArray[4]={1,2,3,4};
for (int i=0;i<4;++i){
    std::cout<<TestArray[i]<<std::endl;
}
```

viene convertito in:

```
int TestArray[4]={1,2,3,4};
for (int i : TestArray){
    std::cout<<i<<std::endl;
}
```

Da notare che se il membro di destra della condizione di terminazione (in questo caso  $i < 4$ ) non fosse stato esattamente uguale alla dimensione dell'array, allora la conversione non sarebbe avvenuta.

## 2.4 modernize-make-shared

Questo check sostituisce ogni occorrenza di `std::shared_ptr` con `std::make_shared`. *Modernize-make-shared* è stato introdotto in quanto il costrutto `std::make_shared` è più performante di `std::shared_ptr`. Infatti il primo effettua un'unica allocazione nello heap, mentre `std::shared_ptr` ne esegue due, il che lo rende anche non sicuro dal punto di vista delle eccezioni.

Consideriamo il seguente esempio:

**[Esempio #2.4]**

```
#include <iostream>
class my_pair{
private:
    int x; int y;
public:
    my_pair(int a, int b);
};
int main(int argc, const char * argv[]) {
    auto my_ptr = std::shared_ptr<my_pair>(new
        my_pair(2,3));
    // Utilizzo del puntatore ...
}
```

Una volta eseguito il comando

**clang-tidy main.cpp -checks="modernize-make-shared" -fix**

ciò che otteniamo è:

```
int main(int argc, const char * argv[]) {
    auto my_ptr = std::make_shared<my_pair>(1, 2);
    // Utilizzo del puntatore ...
}
```

Il discorso è analogo per il check *modernize-make-unique*, che sostituisce ogni occorrenza di *std::unique\_ptr* con *std::make\_unique*.

## 2.5 modernize-pass-by-value

Prima di introdurre il controllo sopracitato, apriamo una parentesi su quella che è stata una grossa novità nello standard **C++11**.

### 2.5.1 Move Semantics

La **Move Semantics** permette di avere del codice più performante evitando la creazione di copie inutili che verranno distrutte.

Consideriamo la seguente porzione di codice:

**[Esempio #2.5.1]**

```
int main(int argc, const char * argv[]) {
    auto a = std::vector<int>(4,100);
    auto b = std::vector<int>(100,4);
    ::swap(a, b);
}
```

Con la funzione *Swap* definita nel seguente modo:

```
1 template <typename T>
2 void swap(T&A, T&B){
3     T TMP(A);
4     A=B;
5     B=TMP;
6 } //Distruzione di TMP
```

All'inizio dell'esecuzione ci troviamo in una situazione in cui A e B puntano agli oggetti con i quali sono stati inizializzati:

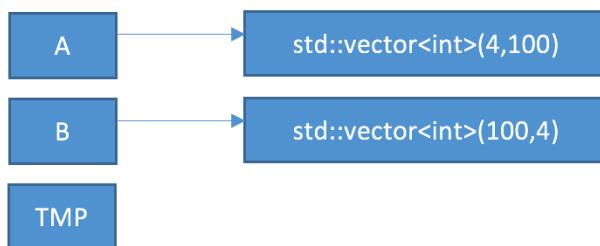


Figura 12: Una copia di A e una copia di B

Alla riga di codice numero 3 la variabile TMP viene inizializzata con una copia di A, quindi dopo l'esecuzione di questa riga ci troviamo con due copie dell'oggetto A e una copia dell'oggetto B.

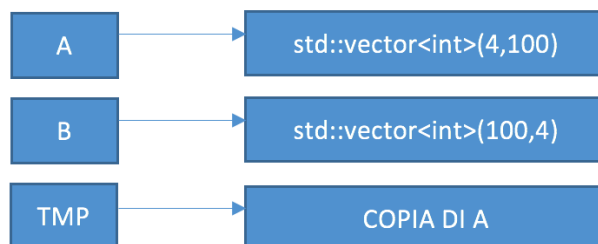


Figura 13: Due copie di A e una copia di B

Dopo l'esecuzione della riga numero 4 ci troviamo ad avere due copie di B e una copia di A.

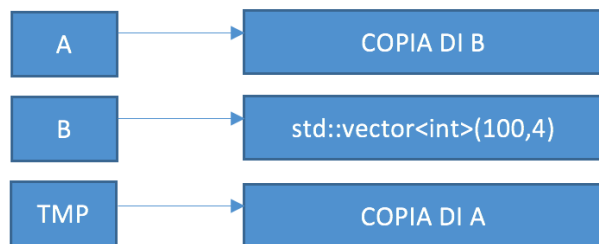


Figura 14: Una copia di A e due copie di B

Alla riga numero 5 avremmo due copie di A e una sola copia di B. Alla fine della funzione otterremo ciò che ci aspettavamo, ovvero che quello che era contenuto in A ora sia in B e viceversa.

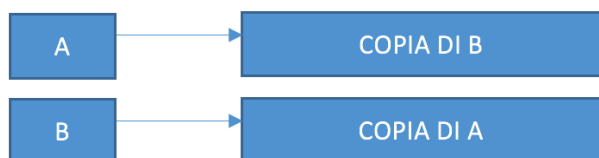
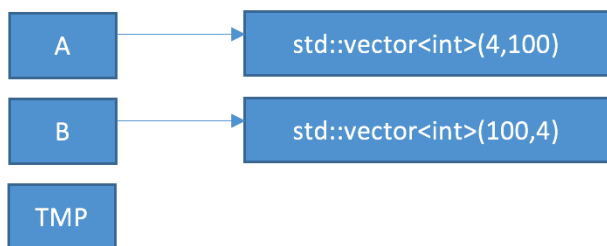


Figura 15: Una copia di A e una copie di B

Per arrivare a tale risultato sono state effettuate ben *tre* copie di vettori, che in questo caso hanno dimensioni ridotte, ma in applicazioni reali potrebbero essere estremamente più grandi. Si può però immaginare una situazione in cui al posto di effettuare delle copie di elementi, questi vengano spostati.

Partiamo dunque dall'esempio precedente:

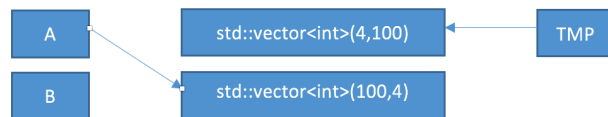


Dopo di che spostiamo lo stato di A in TMP senza eseguire alcuna copia.

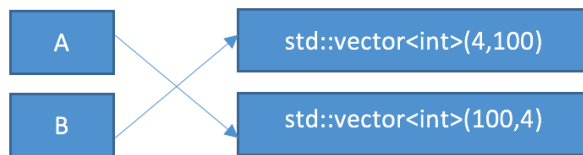




Ora spostiamo lo stato di B in A.



Ed in fine spostiamo lo stato di TMP in A.



Alla fine di tale routine sono state eseguite zero copie, alleggerendo così il carico computazionale e gli accessi in memoria.

Questo tipo di operazione è stata introdotta nello standard *C++11* [9] con il costrutto `std::move()`.

Dunque la funzione `swap` diventerebbe:

```
template <typename T>
void swap(T&A, T&B){
    T TMP(std::move(A));
    A=std::move(B);
    B=std::move(TMP);
} //Distruzione di TMP
```

### 2.5.2 Utilizzo del check

Questo tipo di check permette al compilatore di effettuare la migliore scelta nel creare la copia di un oggetto, sostituendo dove possibile, il passaggio dei parametri per *const-reference* con quello per valore.

Nella versione 4.0 di Clang Tidy, la conversione del passaggio parametri avviene solamente nei costruttori di una classe.

Consideriamo il seguente esempio:

**[Esempio #2.5.2]**

```
class MyClass {
public:
    MyClass(const std::string &Copia, const std::string
            &SolaLettura): Copia(Copia),
            SolaLettura(SolaLettura){};
private:
    std::string Copia;
    const std::string &SolaLettura;
};
std::string get_str; //Parametro costruito con
                    move-constructor.
void f(const std::string &Path) {
    MyClass MyClass(get_str, Path);
}
```

Lanciando da terminale il seguente comando:

```
clang-tidy main.cpp -checks="modernize-pass-by-value" -fix
```

vediamo che Clang-Tidy ha rimosso il *const reference* dal primo parametro sostituendo *Copia(Copia)* con *Copia(std::move(Copia))*.

Di seguito il codice sorgente modificato:

```
class MyClass {
public:
    MyClass(std::string Copia, const std::string
            &SolaLettura): Copia(std::move(Copia)),
            SolaLettura(SolaLettura){};
private:
    std::string Copia;
    const std::string &SolaLettura;
};
```

In tal modo riusciamo ad ottimizzare i tempi di costruzione degli oggetti, rendendo l'applicativo più performante.

## 2.6 modernize-raw-string-literal

Questo check sostituisce i caratteri delimitati in una stringa<sup>13</sup> con i *raw string literals* introdotti nello standard C++11. **[Esempio #2.6]**

```
const char *const Path{"C:\\Vendor\\Application.exe"};
```

Applicandovi Clang-Tidy diventa:

```
const char *const Path{R"(C:Vendor\Application.exe)"};
```

## 2.7 modernize-redundant-void-arg

Questo check trova e rimuove l'utilizzo superfluo di **void** nella lista degli argomenti. **[Esempio #2.7]**

```
int foo(void);  
//Diventa  
int foo();
```

```
C::C(void){};  
//Diventa  
C::C(){};
```

In modo tale da rendere più leggibile il codice sorgente.

## 2.8 modernize-replace-auto-ptr

Replace-auto-ptr sostituisce il tipo *std::auto\_ptr*<sup>14</sup> con *std::unique\_ptr*.<sup>15</sup> Dove invece avviene un trasferimento di proprietà (tramite l'operatore d'assegnamento), esso viene sostituito con una chiamata esplicita di *std::move()*. Consideriamo il seguente esempio: **[Esempio #2.8]**

```
void funzione_con_auto_ptr(std::auto_ptr<int> int_ptr);  
  
int main(int argc, const char * argv[]) {  
    std::auto_ptr<int> a(new int(10));  
    std::auto_ptr<int> b;  
    b = a;  
    funzione_con_auto_ptr(b);  
    return 0;  
}
```

---

<sup>13</sup>Anche noti come *Escaped Characters*

<sup>14</sup>Con lo standard C++11 è stato dichiarato deprecato

<sup>15</sup>Introdotta nello standard C++11

Lanciando Clang Tidy con `checks="modernize-replace-auto-Ptr"` si ottiene il seguente codice sorgente modificato:

```
void funzione_con_auto_ptr(std::unique_ptr<int> int_ptr);  
//Sostituzione di auto_ptr con unique_ptr  
int main(int argc, const char * argv[]) {  
    std::unique_ptr<int> a(new int(10)); //Cambiato in unique_ptr  
    std::unique_ptr<int> b; //Cambiato in unique_ptr  
    b = std::move(a); //Utilizzo esplicito di std::move()  
    funzione_con_auto_ptr(std::move(b));  
    return 0;  
}
```

## 2.9 modernize-shrink-to-fit

La *standard template library* mette a disposizione dei contenitori restringibili<sup>16</sup>: vettori, stringhe e code.

Queste strutture dati hanno due metodi particolari:

- **size()**: che restituisce il numero di elementi all'interno del contenitore.
- **capacity()**: ritorna la capacità massima.

Prima dell'avvento dello standard C++11, per effettuare l'operazione che facesse coincidere il numero di elementi all'interno del contenitore con la capacità massima, si utilizzava l'idioma **Copy and Swap**, che in caso di strutture dati relativamente grandi poteva comportare un calo di prestazioni.

Consideriamo il seguente esempio:

**[Esempio #2.9]**

```
std::vector<int> Vettore(100);  
cout<<"Capacita' del vettore: "<< Vettore.capacity() <<  
    std::endl; // Output: 100  
Vettore.resize(10);  
cout<<"Capacita' del vettore: "<< Vettore.capacity() <<  
    std::endl; //Output: 100  
std::vector<int>(Vettore).swap(Vettore);  
cout<<"Capacita' del vettore: "<< Vettore.capacity() <<  
    std::endl; //Output: 10
```

Il check `modernize-shrink-to-fit` sostituisce la penultima istruzione con il metodo `shrink_to_fit()` che rende più leggibile e comprensibile il codice e in alcuni casi più efficace lasciando l'output invariato.

---

<sup>16</sup>Anche noti come Shrinkable Container

```
std::vector<int> Vettore(100);
cout<<"Capacita' del vettore: "<< Vettore.capacity() <<
    std::endl; // Output: 100
Vettore.resize(10);
cout<<"Capacita' del vettore: "<< Vettore.capacity() <<
    std::endl; //Output: 100
Vettore.shrink_to_fit();
cout<<"Capacita' del vettore: "<< Vettore.capacity() <<
    std::endl; //Output: 10
```

## 2.10 modernize-use-auto

Di seguito uno dei check più utili e interessanti di tutto il pacchetto `modernize`. Tale check rende il codice sorgente più leggibile e anche più manutenibile, soprattutto nei casi in cui si utilizzano gli iteratori.

Consideriamo il seguente esempio: **[Esempio #2.10.1]**

```
std::vector<int>::iterator I = Vettore.begin();
```

viene trasformato in:

```
auto I = Vettore.begin();
```

L'introduzione del tipo `auto` aumenta la leggibilità per quei tipi di dato non built-in, in particolare il check agisce sugli iteratori<sup>17</sup> e sulle *new expression*.

Di seguito alcuni esempi su come tale check aumenti molto la leggibilità del codice sorgente. **[Esempio #2.10.2]**

```
int main(int argc, const char * argv[]) {
    std::vector<int> Vettore(10,10);
    for(std::vector<int>::iterator
        B=Vettore.begin(),E=Vettore.end(); B!=E ; ++B){
        std::cout<<*B<<std::endl;
    }
}
```

---

<sup>17</sup>Iteratori dei contenitore della **STL**. Gli iteratori possono essere **const\_iterator**, **reverse\_iterator**, **const\_reverse\_iterator**, **iterator**.

diventa semplicemente:

```
int main(int argc, const char * argv[]) {
    std::vector<int> Vettore(10,10);
    for(auto B=Vettore.begin(),E=Vettore.end(); B!=E ; ++B){
        std::cout<<*B<<std::endl;
    }
}
```

**[Esempio #2.10.3]**

```
std::vector<int> * Vec_Pointer = new std::vector<int>(10);
```

diventa:

```
auto* Vec_Pointer = new std::vector<int>(10);
```

Per rendere più comprensibile il codice si possono eliminare i puntatori nelle dichiarazioni di variabili aggiungendo al checker la seguente opzione:

```
-config="CheckOptions: [key: modernize-use-auto.RemoveStars,  
value: '1']"
```

restituendo come risultato:

```
auto Vec_Pointer = new std::vector<int>(10);
```

## 2.11 modernize-use-bool-literals

Il seguente controllo sostituisce ogni occorrenza di interi utilizzati per rappresentare booleani con le parole chiave **true** e **false**.

**[Esempio #2.11]**

```
bool Funzione_test(bool x=1){
    bool test1 = 1;
    bool test2 = 0;
    bool test3 = static_cast<bool>(1);
    return 0;
}
```

Eseguido

```
clang-tidy main.cpp -checks="modernize-use-bool-literals" -fix
```

diventa:

```
bool Funzione_test(bool x=true){
    bool test1 = true;
    bool test2 = false;
    bool test3 = true;
    return false;
}
```

## 2.12 modernize-use-default

Questo check rimpiazza la definizione di funzioni membro speciali<sup>18</sup> aventi corpo vuoto, con la parola chiave **default**. **[Esempio #2.12]**

```
class Prova{
    Prova(){} //Costruttore
    ~Prova(){} //Distruttore
    Prova(const Prova&){}; //Costruttore di copia
    Prova(Prova&&); //Move Constructor - Caso non gestito
}
```

viene trasformato in:

```
Prova()= default; //Costruttore
~Prova()= default; //Distruttore
Prova(const Prova&)= default;; //Costruttore di copia
Prova(Prova&&); //Move Constructor - Caso non gestito
```

## 2.13 modernize-use-emplace

Modernize use emplace converte ogni occorrenza del metodo *push\_back()* dei contenitori dell'STL con *emplace\_back()*. Di default agisce solo su *std::vector*, *std::deque*, *std::list*. Questo insieme può essere modificato utilizzando l'opzione *ContainersWithPushBack*.

---

<sup>18</sup>Costruttore, distruttore, costruttore di copia, costruttore di spostamento, costruttore d'assegnamento, e move assignment

## 2.14 modernize-use-nullptr

Il seguente check sostituisce ogni occorrenza di puntatori nulli<sup>19</sup> con la parola chiave **nullptr**. **[Esempio #2.14]**

```
int * Esempio(int* X=0){
    int* a=NULL, *b = 0;
    return 0;
}
```

Viene convertito in:

```
int * Esempio(int* X=nullptr){
    int* a=nullptr, *b = nullptr;
    return nullptr;
}
```

## 2.15 modernize-use-override

Il metodo identifica tutti i metodi che fanno overriding di un metodo virtuale di una classe base e li etichetta con la nuova parola chiave *override*. Facendo così si migliora anche la leggibilità, ma non è questo il vero obiettivo; il vero obiettivo è di evitare quegli errori subdoli dovuti a non avere usato lo stesso identico nome per il metodo, oppure lo stesso numero e tipo di parametri, nel qual caso invece di fare *overriding* si farebbe *hiding*. Si consideri il seguente esempio: **[Esempio #2.15]**

```
class Animali{
public:
    virtual void Verso(){};
};

class Gatto: public Animali{
public:
    virtual void Verso() {
        std::cout<<"Miao";
    }
};
```

Eseguendo il checker otteniamo che al metodo sovrascritto viene tolto l'attributo *virtual* e aggiunto *override*.

---

<sup>19</sup>NULL e 0 (zero)



```
class Gatto: public Animali{
public:
    void Verso() override {
        std::cout<<"Miao";
    }
};
```

In tal modo il codice sorgente ha una maggior leggibilità.

## 2.16 modernize-use-using

Converte ogni occorrenza di *typedef* con *using*.

**[Esempio #2.16]**

```
typedef unsigned char byte;
```

viene trasformato in:

```
using byte = unsigned char;
```

## 3 Estendibilità dell'infrastruttura per i check

Uno degli obiettivi del seguente lavoro era di mettere alla prova la flessibilità e la modularità dell'infrastruttura di Clang Tidy, verificando se un programmatore può realmente adattare il tool ai propri bisogni e con quale sforzo.

Sono stati presi in considerazione tre casistiche:

- La correzione di errori in checker esistenti.
- L'estensione delle funzionalità di checker esistenti.
- La definizione di nuovi checker.

### 3.1 Correzione di un errore in un check esistente

Nello studiare l'efficacia del check *modernize-avoid-bind*, è stato riscontrato un caso per il quale il check *modernize-avoid-bind* proponeva un fix errato (ovvero, una modifica del codice che non preservava il significato). In particolare il check non considerava in alcun modo l'operatore di risoluzione dello scope.<sup>20</sup>

#### 3.1.1 Analisi dell'errore

L'errore è stato trovato testando il check sulla seguente porzione di codice:

```
namespace A {
    int add(int x, int y);
}

int main(int argc, const char * argv[]) {
    auto clj = std::bind(A::add, 1, 1);
    // ...
}
```

dopo avervi applicato il check con l'opzione *-fix*, il risultato ottenuto è stato:

```
int main(int argc, const char * argv[]) {
    auto clj = [=](auto && arg1) { return add(x, arg1); };
    // ...
}
```

Come si vede, nel codice modificato, la funzione *add* è invocata direttamente, senza la necessaria qualificazione con il nome del namespace *A::*). Questo in generale è scorretto, perché potrebbe modificare il risultato della

---

<sup>20</sup>Scope resolution operator '::'

risoluzione dell'*overloading*, causando l'invocazione di un'altra funzione con lo stesso nome. Per essere corretta, la modifica avrebbe dovuto produrre il codice seguente:"

```
int main(int argc, const char * argv[]) {
auto clj = [=](auto && arg1) { return A::add(x, arg1); };
// ...
}
```

si è dunque deciso di procedere con la risoluzione del problema.

### 3.1.2 Analisi del codice sorgente

Per capire dove fosse il problema, è stato necessario analizzare il codice sorgente di tale check, in particolare si nota che il metodo *AvoidBindCheck::registerMatchers(MatchFinder \*Finder)*, considerava solamente la dichiarazione della funzione (*FunctionDecl*), perdendo così tutte le informazioni relative al contesto in cui erano stata invocata. Infatti, nel metodo *AvoidBindCheck::Check( . . . )*, veniva stampato solamente il nome della funzione, confermando dove risiedesse il problema.

Di seguito la definizione delle due funzioni sopra citate:

```
void AvoidBindCheck::registerMatchers(MatchFinder *Finder) {
    if (!getLangOpts().CPlusPlus14) // Need C++14 for generic
        lambdas.
        return;

    Finder->addMatcher(
        callExpr(callee(namedDecl(hasName("::std::bind"))),
            hasArgument(0,
                declRefExpr(to(functionDecl().bind("f"))))
                .bind("bind"),
            this);
    }
}
```

Come si può vedere tale metodo filtra tutte le chiamate a funzione che hanno nome *::std::bind* e come primo argomento un oggetto di tipo *declRefExpr*, etichettando la chiamata a funzione (*CallExpr*) con il nome "bind" ed etichettandone la dichiarazione della funzione (*FunctionDecl*) con il nome "f".

Di seguito il metodo che sfrutta i nodi etichettati in precedenza:

```
void AvoidBindCheck::check(const MatchFinder::MatchResult
    &Result) {
    const auto *MatchedDecl =
        Result.Nodes.getNodeAs<CallExpr>("bind");
```

```
auto Diag = diag(MatchedDecl->getLocStart(), "prefer a
    lambda to std::bind");
// Some Code ...
const auto *F = Result.Nodes.getNodeAs<FunctionDecl>("f");
std::string Buffer;
llvm::raw_string_ostream Stream(Buffer);
// Some Code ...
Stream << "[" << (HasCapturedArgument ? "=" : "") << "]";
addPlaceholderArgs(Args, Stream);
Stream << " { return " << F->getName() << "(";
addFunctionCallArgs(Args, Stream);
Stream << "); }";
Diag <<
    FixItHint::CreateReplacement(MatchedDecl->getSourceRange(),
    Stream.str());
}
```

Come si può vedere dal codice, con l'istruzione `F->getName()`, viene preso in considerazione solamente il nome della funzione e non un eventuale istruzione di scope resolution.

### 3.1.3 Risoluzione

L'idea per procedere con la risoluzione dell'errore è stata quella di etichettare non solo le *FunctionDecl* e le *CallExpr* ma anche le *DeclRefExpr*, che rappresentano il riferimento ad una dichiarazione (in questo caso la dichiarazione di una funzione). In tal modo, è possibile prendere non solo il nome della funzione ma anche il contesto nella quale è stata dichiarata, avendo tutte le informazioni su un eventuale operatore di scope resolution.

Di seguito, la modifica effettuata sul metodo `AvoidBindCheck::registerMatchers`, dove ogni nodo di tipo *DeclRefExpr* viene etichettato con il nome "ref".

```
void AvoidBindCheck::registerMatchers(MatchFinder *Finder) {
    if (!getLangOpts().CPlusPlus14) // Need C++14 for generic
        lambdas.
    return;

    Finder->addMatcher(
        callExpr(callee(namedDecl(hasName("::std::bind"))),
            hasArgument(0,
                declRefExpr(to(functionDecl().bind("f")).bind("ref")))
                .bind("bind")),
        this); }
```

Dopodiché abbiamo proceduto, con l'ottenere il nome corretto della funzione utilizzando il nodo **ref**.<sup>21</sup>

```
void AvoidBindCheck::check(const MatchFinder::MatchResult
    &Result) {
const auto *MatchedDecl =
    Result.Nodes.getNodeAs<CallExpr>("bind");
auto Diag = diag(MatchedDecl->getLocStart(), "prefer a lambda
    to std::bind");
// Some Code ...
const auto *F = Result.Nodes.getNodeAs<FunctionDecl>("f");
std::string Buffer;
llvm::raw_string_ostream Stream(Buffer);
// Some Code ...
const auto *ref = Result.Nodes.getNodeAs<DeclRefExpr>("ref");
Stream << "[" << (HasCapturedArgument ? "=" : "") << "]";
addPlaceholderArgs(Args, Stream);
Stream << " return ";
ref->printPretty(Stream, nullptr,
    Result.Context->getPrintingPolicy());
Stream << "(";
addFunctionCallArgs(Args, Stream);
Stream << "); }";

Diag <<
    FixItHint::CreateReplacement(MatchedDecl->getSourceRange(),
    Stream.str());
}
```

### 3.1.4 Testing

Prima di rilasciare la patch, è stato necessario testare che le modifiche apportate al Check, non abbiano aggiunto ulteriori errori e che il checker nel suo insieme fosse ancora stabile.

*LLVM* dispone di una suite di test. In particolari quelli del pacchetto *modernize* sono locati nella directory */llvm/tools/clang/tools/extra/test/clang-tidy*.

---

<sup>21</sup>In verde, il codice aggiunto, in arancione quello alla quale sono state apportate delle modifiche.

Nel file `modernize-avoid-bind.cpp` sono state aggiunte le seguenti istruzioni:

```
namespace C {
  int add(int x, int y){ return x + y; }
}
void n(){
  auto clj = std::bind(C::add, 1, 1);
  // CHECK-MESSAGES: :[[@LINE-1]]:16: warning: prefer a lambda
  // to std::bind
  // CHECK-FIXES: auto clj = [] { return C::add(1, 1); };
}
```

I test falliscono se non viene stampato il *CHECK-MESSAGES* e se dopo l'esecuzione del checker le modifiche apportate sono diverse da *CHECK-FIXES*. I test sono stati lanciati con il seguente comando:

#### **llvm-lit modernize-avoid-bind.cpp**

ottenendo:

```
-- Testing: 1 tests, 1 threads -- PASS: Clang Tools ::
  clang-tidy/modernize-avoid-bind.cpp (1 of 1)
Testing Time: 0.61s
Expected Passes : 1
```

Per avere una contro prova, sono state rimosse le modifiche apportate nel file *AvoidBindCheck.cpp* lasciando però invariato il file dei test, ed quello che otteniamo rilanciando *llvm-lit* è il seguente errore:

```
-- Testing: 1 tests, 1 threads --
FAIL: Clang Tools :: clang-tidy/modernize-avoid-bind.cpp (1 of
  1)
Testing Time: 0.62s
*****
Failing Tests (1):
Clang Tools :: clang-tidy/modernize-avoid-bind.cpp

Unexpected Failures: 1
```

Ciò ha dimostrato che la risoluzione del bug è corretta.

### 3.1.5 Patch

Per poter rendere effettive le modifiche attutate, è stato necessario proporre la patch alla community di LLVM, passando attraverso la mailing list *cfe-commits*. Per poter creare il file contenente la patch, si è prima dovuto allineare la versione della repository sulla quale stavamo lavorando<sup>22</sup>, utilizzando il comando *svn update*, dopo di che sono state create le patch con il comando *diff*, che è stato applicato sul file dei test e sulla classe *avoid-bind*.

Di seguito le due patch:

```
Index: AvoidBindCheck.cpp
=====
--- AvoidBindCheck.cpp (revision 282328)
+++ AvoidBindCheck.cpp (working copy)
@@ -109,7 +109,7 @@

Finder->addMatcher(
callExpr(callee(namedDecl(hasName("::std::bind"))),
-         hasArgument(0,
+         declRefExpr(to(functionDecl().bind("f")))))
+         hasArgument(0,
+         declRefExpr(to(functionDecl().bind("f")).bind("ref")))
.bind("bind"),
this);
}
@@ -148,10 +148,13 @@

bool HasCapturedArgument = llvm::any_of(
Args, [](const BindArgument &B) { return B.Kind == BK_Other; });
+ const auto *ref = Result.Nodes.getNodeAs<DeclRefExpr>("ref");

Stream << "[" << (HasCapturedArgument ? "=" : "") << "];
addPlaceholderArgs(Args, Stream);
- Stream << " { return " << F->getName() << "(";
+ Stream << " { return ";
+ ref->printPretty(Stream, nullptr,
+   Result.Context->getPrintingPolicy());
+ Stream<< "(";
addFunctionCallArgs(Args, Stream);
Stream << "); }";
```

Figura 16: Patch del File: AvoidBindCheck.cpp

---

<sup>22</sup>Nota anche come working copy.

```
Index: modernize-avoid-bind.cpp
=====
--- modernize-avoid-bind.cpp      (revision 280344)
+++ modernize-avoid-bind.cpp      (working copy)
@@ -68,3 +68,12 @@
 // CHECK-FIXES: auto clj = std::bind(add, 1, add(2, 5));
 }

+namespace C {
+ int add(int x, int y){ return x + y; }
+}
+
+void n(){
+ auto clj = std::bind(C::add, 1, 1);
+ // CHECK-MESSAGES: :[@LINE-1]:16: warning: prefer a lambda to
+   std::bind
+ // CHECK-FIXES: auto clj = [] { return C::add(1, 1); };
+}
```


Figura 17: Patch del File: modernize-avoid-bind.cpp

Alla mailing list vengono inoltrate tutte le attività effettuate su *Phabricator*.<sup>23</sup> Una revisione è composta da un autore e da uno o più revisori. Il compito dei revisori è quello di controllare e testare la patch e dunque accettarla o rigettarla. La patch proposta è stata accettata da due revisori, e dunque è stato possibile procedere con il commit delle modifiche.

---

<sup>23</sup>Un insieme di strumenti per gli sviluppatori di software che comprende la gestione dei tasks, degli sprint, della revisione del codice e delle repository quali git, svn o mercurial.



 Authored by **IdrissRio** on Sun, Oct 16, 12:58 AM.

#### Details

Reviewers  alexfh  
 aaron.ballman  
 hokein

---

#### ☰ SUMMARY

Hello, i would like to suggest a fix for one of the checks in clang-tidy and i should hope this one is the correct mailing list. The check is modernize-avoid-bind.

Consider the following:

```
void bar(int x, int y);

namespace N {
    void bar(int x, int y);
}

void foo(){
    auto Test = std::bind(N::bar,1,1);
}
```

clang-tidy's modernize-avoid-bind check suggests writing:

```
void foo(){
    auto Test = [] {return bar(1,1);};
}
```

instead of:

```
void foo(){
    auto Test = [] {return N::bar(1,1);};
}
```

So clang-tidy has proposed an incorrect Fix.

Figura 18: Link della revisione: <https://reviews.llvm.org/D25649>

### 3.2 Estensione di un check: Modernize Use Default

Per determinate necessità è stato necessario modificare il comportamento del check *modernize-use-default* sui costruttori definiti *inline*. Infatti in presenza di tali costruttori Clang Tidy suggerisce di aggiungere `"=default"` nella definizione del metodo e non sulla dichiarazione.<sup>24</sup>

Per comprenderne meglio il comportamento, consideriamo il seguente esempio: **[Esempio #3.2.1]**

```
template <typename U>
class A{
    A<U>();
    A<U>(A<U>& u);
};
#include "inline.hh"
```

con la seguente implementazione dei metodi nel file *inline.hh*:

```
template <typename U>
inline A<U>::A(){}

template <typename U>
inline A<U>::A(A<U>& u){}
```

eseguendo il checker su tali sorgenti, si ottiene il seguente risultato:

```
template <typename U>
inline A<U>::A()=default;

template <typename U>
inline A<U>::A(A<U>& u)=default;
```

anziché

```
template <typename U>
class A{
    A<U>()=default;
    A<U>(A<U>& u)=default;
};
#include "inline.hh"
```

Rimuovendo completamente le implementazioni dei costruttori dal file *inline.hh*.

---

<sup>24</sup> con relativa eliminazione della definizione.

### 3.2.1 Implementazione

Per implementare il nuovo comportamento del check, si è dovuto modificare il file *UseDefaultCheck.cpp* dove sono definiti i metodi della classe. In particolare è stato modificato il metodo *UseDefaultCheck::Check*, cercando le funzioni definite *out of line*<sup>25</sup> e *inline*.

Di seguito la porzione di codice modificata:

```
const auto *SpecialFunctionDecl =
Result.Nodes.getNodeAs<CXMethodDecl>(SpecialFunction);
//Some Code - Inizialization of SpecialFunctionName
if(SpecialFunctionDecl->isOutOfLine() && SpecialFunctionDecl
-> isInlineSpecified()){
    const auto *SpecialFunctionFirstDeclaration =
        SpecialFunctionDecl->getFirstDecl();
    diag(SpecialFunctionDecl->getLocStart(), "Remove inline
definition and set '= default' in the declaration") <<
        FixItHint::CreateRemoval(
            CharSourceRange::getTokenRange(
                SpecialFunctionDecl->getLocStart(),
                SpecialFunctionDecl->getLocEnd()));

    PresumedLoc PLoc=Result.SourceManager-> getPresumedLoc(
        SpecialFunctionFirstDeclaration->getLocEnd());
    auto NewLocation = Result.SourceManager->
        translateLineCol(Result.SourceManager->
            getFileID(SpecialFunctionFirstDeclaration->
                getLocEnd()), PLoc.getLine(), PLoc.getColumn()+1);
    diag(SpecialFunctionFirstDeclaration-> getLocEnd(), "use '=
default' to define a trivial " + SpecialFunctionName)<<
        FixItHint::CreateInsertion(NewLocation, " = default");
    return;
}
diag(SpecialFunctionDecl->getLocStart(),
"use '= default' to define a trivial " + SpecialFunctionName)
<<FixItHint::CreateReplacement(
    CharSourceRange::getTokenRange(StartLoc, EndLoc), "=
default;");
}
}
```

---

<sup>25</sup>Funzioni definite in un'unità di traduzione diversa da quella della dichiarazione.

Dopo avere trovato le definizioni out of line e inline (*SpecialFunctionDecl*), è stata ricavata la *prima dichiarazione* (*SpecialFunctionFirstDeclaration*), utilizzando la funzione *getFirstDecl()*. Dopo di che, con il metodo *FixItHint::CreateRemoval* è stata eliminata la definizione del costruttore, mentre con *FixItHint::CreateInsertion*, è stato aggiunto "**=default**" alla dichiarazione.

### 3.3 Creazione di un check: Modernize Use Delete

Clang-tidy gode di una forte modularità; questo permette, a chiunque abbia un'idea per la realizzazione di un nuovo check, di implementarla e testarla senza troppe complicazioni.

Prima dell'avvento dello standard del *C++11*, per non permettere l'utilizzo di costruttori/distruttori di una classe, li si definiva *privati* e *non implementati*. Dichiarandoli privati se ne impedisce l'uso da parte del codice "esterno" alla classe stessa (le funzioni non membro che non sono state dichiarate *friend* della classe), che nel caso di invocazione otterrebbe un errore di compilazione. L'assenza dell'implementazione, invece, serve ad impedirne l'uso da parte del codice "interno" alla classe (le funzioni membro e le funzioni dichiarate *friend* della classe): in questo caso, l'invocazione provoca un errore in fase di collegamento.

Con l'introduzione del nuovo standard, è stata aggiunta la keyword '*delete*', il cui scopo è quello di non permettere l'utilizzo dei costruttori/distruttori da parte degli utenti della classe.

Il seguente esempio mostra come l'utilizzo della keyword '*delete*', renda il codice sorgente più autoesplicativo. Di seguito, la definizione della classe *MyClass* prima della pubblicazione dello standard *C++11*.

#### [Esempio #3.3.1]

---

```
class MyClass{
private:
    MyClass(const MyClass&);
    MyClass& operator=(const MyClass&);
};
```

---

Mentre la porzione di codice a seguire rappresenta la definizione della stessa classe con l'utilizzo dello standard *C++11*.

---

```
class MyClass{
public:
    MyClass(const MyClass&) = delete;
    MyClass& operator=(const MyClass&) = delete;
};
```

---

### 3.3.1 Implementazione

Per poter creare un nuovo check da poter poi eseguire utilizzando Clang-Tidy, è stato necessario creare due file: *UseDeleteCheck.cpp* e il relativo header file *UseDeleteCheck.h*. Nell'header file sono stati dichiarati i metodi già descritti nel capitolo Struttura di un check. Di seguito il file *UseDeleteCheck.h*:

```
#ifndef
    LLVM_CLANG_TOOLS_EXTRA_CLANG_TIDY_MODERNIZE_USE_DELETE_H
#define
    LLVM_CLANG_TOOLS_EXTRA_CLANG_TIDY_MODERNIZE_USE_DELETE_H

#include "../ClangTidy.h"
namespace clang {
namespace tidy {
namespace modernize {

class UseDeleteCheck : public ClangTidyCheck{
public:
    UseDeleteCheck(StringRef Name, ClangTidyContext
        *Context):ClangTidyCheck(Name, Context) {}
    void registerMatchers(ast_matchers::MatchFinder
        *Finder) override;
    void check(const
        ast_matchers::MatchFinder::MatchResult &Result)
        override;
}; // Class UseDeleteCheck

} // namespace modernize
} // namespace tidy
} // namespace clang
#endif //LCM_CLANG_TOOLS_EXTRA_CLANG_TIDY_MODERNIZE_USE_DELETE_H
```

Di seguito l'overriding del metodo *registerMatchers*:

```
static const char SpecialFunction[] = "SpecialFunctionBind";

void UseDeleteCheck::registerMatchers(MatchFinder *Finder) {
    if (getLangOpts().Cplusplus) {
        // Distruttore
        Finder->addMatcher(cxxDestructorDecl
            (isPrivate()).bind(SpecialFunction), this);
        Finder->addMatcher(
            cxxConstructorDecl(
```

```
    anyOf(
      // Costruttore di default
      allOf(isPrivate(), unless(
        hasAnyConstructorInitializer(anything()),
        parameterCountIs(0)),
      // Costruttore di copia
      allOf(
        isPrivate(), isCopyConstructor(),
        parameterCountIs(1)))
    .bind(SpecialFunction),
    this);
  // Operator=
  Finder->addMatcher(
    cxxMethodDecl(isPrivate(), isCopyAssignmentOperator(),
      hasParameter(0, hasType(lValueReferenceType())))
    .bind(SpecialFunction),
    this);
}
}
```

Con il metodo *registerMatchers* sono stati etichettati tutti i nodi dell'AST che corrispondessero a dei costruttori/distruttori privati, con il nome *SpecialFunctionBind*. Dopo aver visitato tutto l'AST ed etichettato tutti i nodi, con il metodo *UseDeleteCheck::check*, sono stati filtrati tutti i costruttori/distruttori che non fossero definiti dall'utente o già marcati con *=delete*/*=default*.

Dopo aver determinato quale tipo di costruttore/distruttore fosse<sup>26</sup>, con la funzione *FixItHint::CreateInsertion*, è stata aggiunta la keyword *=delete*. Di seguito l'implementazione del metodo *UseDeleteCheck::check*:

```
void UseDeleteCheck::check(const MatchFinder::MatchResult
  &Result) {
  std::string SpecialFunctionName;
  const auto *SpecialFunctionDecl =
    Result.Nodes.getNodeAs<CXXMethodDecl>(SpecialFunction);

  if (SpecialFunctionDecl->isDefined() ||
      SpecialFunctionDecl->isDeleted() ||
      SpecialFunctionDecl->isExplicitlyDefaulted() ||
      SpecialFunctionDecl->isLateTemplateParsed() ||
      !SpecialFunctionDecl->isUserProvided() )
    return;
```

---

<sup>26</sup>"default constructor", "copy constructor", "destructor", "copy-assignment operator"

```
if (const auto *Ctor =
    dyn_cast<CXXConstructorDecl>(SpecialFunctionDecl)) {
    if (Ctor->getNumParams() == 0) {
        SpecialFunctionName = "default constructor";
    } else {
        SpecialFunctionName = "copy constructor";
    }
} else if (isa<CXXDestructorDecl>(SpecialFunctionDecl)) {
    SpecialFunctionName = "destructor";
} else {
    SpecialFunctionName = "copy-assignment operator";
}
PresumedLoc PLoc = Result.SourceManager->
    getPresumedLoc(SpecialFunctionDecl->getLocEnd());
auto NewLocation = Result.SourceManager->
    translateLineCol(Result.SourceManager->
        getFileID(SpecialFunctionDecl->getLocEnd()),
        PLoc.getLine(), PLoc.getColumn()+1);
diag(NewLocation, "Use '= delete' to disable the " +
    SpecialFunctionName)<<
    FixItHint::CreateInsertion(NewLocation, " = delete");
}
```

Dopo aver concluso la fase d'implementazione del check, è stato necessario aggiungere una relazione tra il nome della classe e la stringa che verrà poi utilizzata per invocare il check da Clang-Tidy. Per fare ciò si è modificato il file **ModernizeTidyModule.cpp** aggiungendo al metodo *addCheckFactories* la seguente riga di codice:

```
CheckFactories.registerCheck<UseDeleteCheck>("modernize-use-delete");
```

In tal modo il check è utilizzabile lanciando il comando:

```
clang-tidy SomeFile.cpp -checks="modernize-use-delete"
```

### 3.4 Modernize Use Enum Class

Prima della pubblicazione dello standard *C++11*, a causa di un conflitto di nomi, non si potevano utilizzare enumerazioni con almeno una dichiarazione in comune, come nel seguente esempio: **[Esempio #3.4.1]**

```
enum TrafficLight {
    green,
    orange,
    red
};

enum RGB {
    red,
    green,
    blue
};
```

Infatti, tale porzione di codice non viene compilata in quanto *red* risulta essere in ambedue le enumerazioni. Per risolvere il conflitto dei nomi, l'enumerazione veniva dichiarata all'interno di un *namespace* creato unicamente per ovviare tale problema. La sintassi utilizzata era la seguente:

```
namespace TrafficLight {
    enum { green, orange, red };
}

namespace RGB {
    enum { red, green, blue };
}
```

Richiamando gli elementi all'interno delle enumerazioni nel seguente modo:

```
TrafficLight::red;
RGB::red;
```

Per evitare di dover utilizzare una sintassi poco chiara per definire un'enumerazione, nello standard *C++11* sono state introdotte le **enum class** il cui scopo è quello di poter utilizzare le enumerazioni come sopra, evitando l'utilizzo dei *namespace*.

```
enum class TrafficLight {green, orange, red};

enum class RGB { red, green, blue};
```

Rendendo il codice sorgente più ordinato e leggibile.



Si è dunque deciso di creare un check che per ogni *namespace* con un'unica dichiarazione di enumerazione effettuasse la modifica sostituendola con una *enum class*.

#### 3.4.1 Implementazione

Così come per il check *Modernize Use Delete* anche qui sono stati creati due file: *ReplaceNamespaceClass.cpp* e il relativo header file *ReplaceNamespaceClass.hh*.

Di seguito l'header file con la dichiarazione dei metodi utilizzati:

```
#ifndef
LLVM_CLANG_TOOLS_EXTRA_CLANG_TIDY_MODERNIZE_REPLACE_NAMESPACE_CLASS_H
#define
LLVM_CLANG_TOOLS_EXTRA_CLANG_TIDY_MODERNIZE_REPLACE_NAMESPACE_CLASS_H

#include "../ClangTidy.h"
namespace clang {
namespace tidy {
namespace modernize {

class ReplaceNamespaceClass : public ClangTidyCheck{
public:

ReplaceNamespaceClass(StringRef Name, ClangTidyContext *Context)
: ClangTidyCheck(Name, Context) {}
void registerMatchers(ast_matchers::MatchFinder *Finder) override;
void check(const ast_matchers::MatchFinder::MatchResult &Result)
override;
}; // Class ReplaceNamespaceClass

} // namespace modernize
} // namespace tidy
} // namespace clang
#endif
//LLVM_CLANG_TOOLS_EXTRA_CLANG_TIDY_MODERNIZE_REPLACE_NAMESPACE_CLASS_H
```

Il metodo *registerMatchers* è stato implementato in modo tale da etichettare tutti e soli i nodi dell'AST che rappresentassero una *namespaceDecl*. Mentre il metodo *ReplaceNamespaceClass::check* è stato studiato in maniera tale da ricavare da ogni *namespace* la chiusura superiore, raccogliendo tutte le dichiarazioni presenti nell'unità di traduzione analizzata.

Si consideri il seguente esempio:

**[Esempio #3.4.2]**

```
//File X.hh
namespace N {
int x;
}
//File Y.hh
namespace N {
int y;
}
```

risulta essere equivalente a:

```
namespace N {
int x;
int y;
}
```

Una volta ricavata la chiusura il metodo esclude ogni *namespace* con più di una dichiarazione, mentre per i restanti controlla che l'unica presente sia un'enumerazione. Dopodiché, se attivata l'opzione *-fix*, il metodo procede creando un *enum class* con nome del *namespace* e corpo dell'enumerazione.

Di seguito il sorgente relativo al metodo *ReplaceNamespaceClass::check*:

```
void ReplaceNamespaceClass::check(const MatchFinder::MatchResult
&Result) {
    const NamespaceDecl *NSD =
        Result.Nodes.getNodeAs<NamespaceDecl>(NameSpaceDeclaration);
    const auto UpperNameSpaceDecl = NSD->getEnclosingNamespaceContext();
    if(UpperNameSpaceDecl->decls_empty())
        return;
    //FIX-IT:se ci sono commenti stile doxygen quindi /// comment; /**
    Comment */; //! Comment non effettuiamo alcun tipo di modifica.
    short count=0;
    auto it = UpperNameSpaceDecl->decls_begin();
    for(;it!=UpperNameSpaceDecl->decls_end();++it){
        ++count;
        if(count>1)
            return;
    }
    it=UpperNameSpaceDecl->decls_begin();
    //Se la dichiarazione e' una EnumDecl allora possiamo procedere
    if(auto enumDecl = dyn_cast<EnumDecl>(*it)){
        std::string NSDName = NSD->getNameAsString();
        std::string corpo =
            Lexer::getSourceText(CharSourceRange::getTokenRange
            (enumDecl->getBraceRange()),*Result.SourceManager,
            Result.Context->getLangOpts());
```

```
diag(NSD->getLocStart(),"Replace 'namespace' scope resolution  
with 'scoped enumeration'") << FixItHint::CreateReplacement(  
CharSourceRange::getTokenRange  
(NSD->getLocStart(),NSD->getLocEnd()), "enum class "+ NSDName  
+" " + corpo + ";");  
}  
}
```

In fine per poter utilizzare il metodo lo si è registrato nel file *ModernizeTidyModule.cpp*, aggiungendo alla funzione *addCheckFactories* la seguente porzione di codice:

```
CheckFactories.registerCheck<ReplaceNamespaceClass>  
("modernize-use-enum-class");
```

Per compilare *Clang Tidy* è stato utilizzato il comando *make -j 4*. Una volta terminato il processo di compilazione è stato possibile utilizzare il check lanciando il seguente comando:

```
clang-tidy SomeFile.cpp -checks="modernize-use-enum-class");
```

## 4 Testing su PPL

Per poter valutare l'effettiva efficacia di *Clang Tidy Modernize* è stato necessario testarlo su un progetto concreto. E' stato scelto di testare i singoli check sulla versione 1.2 della *Parma Polyhedra Library*, controllando dove questi sbagliassero o facessero la modifica corretta, studiando così le limitazioni dei modernizzatori automatici.

### 4.1 Parma Polyhedra Library

La **Parma Polyhedra Library** [10], d'ora in avanti *PPL*, è una libreria di astrazioni numeriche concepita per le applicazioni nel campo dell'analisi e verifica di sistemi complessi. Queste astrazioni includono:

- Poliedri Complessi: definiti come l'intersezione di un numero finito di semispazi (aperti o chiusi), ciascuno descritto da una disuguaglianza lineare (stretta o non stretta) con coefficienti razionali.
- Classi speciali di poliedri con lo scopo di offrire un compromesso tra complessità e precisione.
- Grid: che rappresentano punti regolarmente distribuiti che soddisfano un insieme di relazioni di congruenza lineari.

La libreria, inoltre, offre supporto per:

- Insiemi delle parti finiti di poliedri e prodotti cartesiani tra (ogni tipo di) poliedri e grids.
- Un risolutore ad aritmetica esatta, utilizzando l'algoritmo del simplesso, per la risoluzione di problemi di programmazione lineare a interi misti.
- Un risolutore per la programmazione lineare parametrica.
- Primitive per l'analisi di terminazione via sintesi automatica di funzioni lineari di ranking.

La *PPL* è composta da 78 implementation file <sup>27</sup> e 366 header file.<sup>28</sup>

### 4.2 Preparazione dell'ambiente di testing

*Clang Tidy* è stato progettato per essere eseguito sequenzialmente sui file d'implementazione, analizzando di conseguenza *solamente* gli header file inclusi. Questo porta ad una perdita d'informazioni, come ad esempio la chiusura superiore completa di un namespace.

E' stato dunque deciso di eseguire *Clang Tidy* contemporaneamente su ogni file implementativo attraverso il seguente script:

```
allFile=""
while read p; do
  all+=" $p"
done<PPLAllFileDotCC.txt
# $all e' la stringa contenente tutti i file .cc della PPL
clang-tidy $all -checks="modernize-use-delete" -header-filter=.* -fix
  -fix-errors -- -std=c++11
```

Il risultato ottenuto è stato inaspettato in quanto *Clang Tidy*, prima processa tutti i file, creando un lista di modifiche e solo successivamente le applica [11]. Ottenendo risultati come questi:

```
class Free_List{
//Some Methods ...
private:
  Free_List(const Free_List&()); // Not Implemented.
}
```

trasformandolo in

```
class Free_List{
// Some Methods ...
Private:
```

---

<sup>27</sup>Con estensione *.cc*

<sup>28</sup>Con estensione *.hh*

```
F = delete = delete = delete = delete = delete = delete
    = delete = delete = delete = delete = delete = delete =
    delete = delete = delete = delete = delete = delete =
    delete = delete = delete = delete = delete = delete =
    delete = delete = delete = delete = delete = delete =
    delete = delete = delete = delete =
    deleteree_List(const Free_List&) = delete = delete =
    delete = delete = delete = delete = delete = delete =
    delete = delete = delete = delete = delete = delete =
    delete = delete = delete = delete = delete = delete =
    delete = delete = delete = delete = delete = delete =
    delete = delete = delete = delete = delete = delete =
    delete; // Not Implemented
}
```

La soluzione proposta dagli sviluppatori di Clang Tidy è quella di utilizzare lo script **run-clang-tidy.py** che:

1. Parallelizza diverse istanze di Clang Tidy.
2. Tiene traccia dei cambiamenti da applicare (utilizzando l'opzione "clang-tidy -export-fixes").
3. Solo quando il processo d'analisi dei file termina, vengono applicati i cambiamenti (utilizzando l'eseguibile "clang-apply-replacements")

Per poter utilizzare *run-clang-tidy.py* è stato necessario definire per ogni file d'implementazione le opzioni di compilazione.

Con il seguente script è stato generato il file **compile\_commands.json**:

```
all=""
while read p; do
all+='{"directory":
    "/Users/idriss/Desktop/OneDrive/UNIPR/CLANG/Tesi/PPL/src/",
    "command": "clang++ -std=c++11 -std=c++11 -c '$p' -o", "file":
    "'$p' " }, \n'
done<PPLAllFileDotCC.txt
all+="]"
echo $all >> compile_commands.json
```

Lanciando dunque il comando:

```
run-clang-tidy.py -checks="modernize-use-default" -header-filter=.*
-p=$PWD -fix -j 4
```

si è riusciti ad eseguire Clang Tidy su tutti i file d'implementazione senza perdere alcuna informazione ed evitando i *fix* ripetuti.

### 4.3 Statistiche

Dopo avere preparato l'ambiente di testing si è potuto iniziare ad eseguire Clang Tidy su tutti i file d'implementazione.

Il controllo sull'effettiva correttezza<sup>29</sup> delle modifiche apportate dal checker è avvenuto secondo due modalità differenti:

- Automatica: ricompilando l'intera PPL, assicurandosi che il checker non introduca errori di compilazione.
- Visiva: tramite software di versionamento, confrontando manualmente la *working copy* con la copia originale della PPL.

A seguire i risultati<sup>30</sup> ottenuti da ogni check eseguito:

Check Name	Fix Corretti	Fix Errati
modernize-use-nullptr	247	0
modernize-use-auto	224	0
modernize-use-override	114	0
modernize-use-default	58	0
modernize-pass-by-value	19	2
modernize-use-delete	28	0
modernize-use-emplace	15	0
modernize-replace-auto-ptr	2	0
modernize-redundant-void-arg	2	0
modernize-deprecated-headers	1	0

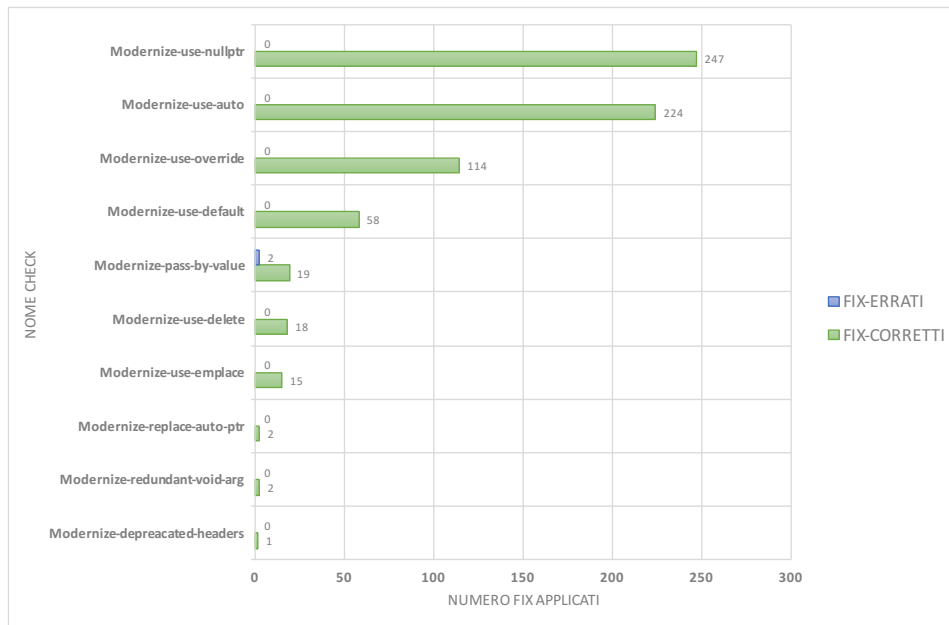
---

<sup>29</sup>Non è stato possibile effettuare un controllo completo sulla correttezza della semantica della PPL dopo l'applicazione di Clang Tidy, in quanto in generale l'equivalenza di due programmi è proprietà indecidibile.

<sup>30</sup>Sono stati riportati solo i check che hanno eseguito almeno un fix.

## 4 TESTING SU PPL

---



Come si può vedere dai dati ottenuti il risultato ha confermato ciò che si aspettava, ovvero che il numero di fix applicati è molto ristretto rispetto alle grandi dimensioni della PPL: questo perché essa è stata scritta seguendo lo standard *C++98* ed utilizzando uno stile di programmazione avanzata, mentre Clang Tidy agisce soprattutto su porzioni di codice relativamente semplici.

Analizziamo più nel dettaglio i *fix* ed i problemi riscontrati durante l'applicazione:

- **modernize-use-default:** sono state riscontrate 58 situazioni nella quale si poteva utilizzare la keyword '= delete' di cui 49 erano definizioni *'inline'* e dunque le modifiche sono state applicate come spiegato nel capitolo 3.2.
- **modernize-use-delete:** nel testare il seguente test, sono stati riscontrati problemi a livello dell'AST. In particolare nel matcher venivano registrate anche le dichiarazioni di costruttori/distruttori che corrispondevano ad istanze di template. La soluzione poteva avvenire sia a livello del matcher, che a livello del check. Per questioni d'ottimizzazione di spazio in memoria si è deciso di agire a livello *matcher*, aggiungendo ad ogni *addMatcher* la condizione *unless(ast\_matchers::isTemplateInstantiation())*.
- **modernize-pass-by-value:** dopo l'applicazione di tale check sull'intera PPL, sono stati riscontrati due errori di compilazione. In parti-



colare il check ha modificato le dichiarazioni di alcuni metodi senza modificare la relativa definizione.

Di seguito gli errori riscontrati:

```
#!/ File iterator_to_const_defs.hh:135
template <typename Container>
class Parma_Polyhedra_Library::const_iterator_to_const {
private:
    ...
    /*! Constructs from the lower-level const_iterator.
    const_iterator_to_const(const Base& b);
};
```

Correttamente diventa

```
#!/ File iterator_to_const_defs.hh:135
template <typename Container>
class Parma_Polyhedra_Library::const_iterator_to_const {
private:
    ...
    /*! Constructs from the lower-level const_iterator.
    const_iterator_to_const(Base b);
};
```

Lasciando però invariata la testata della definizione

```
#!/ File iterator_to_const_inline.hh:119
template <typename Container>
inline
const_iterator_to_const<Container>::
    const_iterator_to_const(const Base& b)
: base(std::move(b)) {
}
```

Lo stesso errore si è presentato per il costruttore di copia della classe *iterator\_to\_const*.

I restanti check non hanno presentato problemi durante l'applicazione. Però come si può vedere, scartando i check *modernize-replace-auto-ptr*, *modernize-redundant-void-arg* e *modernize-deprecated-headers*<sup>31</sup>, solo sette su diciannove si sono rilevati utili.

---

<sup>31</sup>Che hanno effettuato un numero irrilevante di fix.

## 5 Conclusioni

Il presente lavoro di tesi ha permesso di studiare più a fondo le problematiche relative alla modernizzazione del codice sorgente mediante checker automatici.

In particolare *Clang Tidy Modernize* si è presentato come un interessante progetto open source, ben strutturato la cui proprietà fondamentale è la modularità, permettendo a chiunque di creare nuovi check.

I problemi principali riscontrati durante l'utilizzo di Clang Tidy sono i fix ripetuti e alcuni errori nei checker. Il problema dei fix ripetuti è principalmente dovuto alla struttura con la quale è stato pensato Clang Tidy e le principali cause sono:

- Espansioni di macro/template: in quanto il codice scritto nel corpo di un'espansione può avere significati diversi a seconda dell'espansione della macro. Lo stesso problema si pone per le istanze di template, che nell'AST vengono gestite in maniera totalmente differente.
- *Header* file inclusi in più unità di traduzione.

Il problema dei fix ripetuti è in parte risolto dallo script in Python *run-clang-tidy.py*, difficile da configurare e dunque orientato agli utenti più esperti.

Nel complesso Clang Tidy si è presentato come un ottimo tool di supporto per il programmatore nell'attività di modernizzazione del codice C++, anche se tuttora non è affidabile ed efficace come alcuni software proprietari già esistenti.

### 5.1 Sviluppi futuri

Nel futuro si continuerà a contribuire allo sviluppo di Clang Tidy, proponendo nuovi check e risolvendo errori nel checker, come nel caso di *modernize-pass-by-value*. Considerata l'elevata difficoltà iniziale riscontrata nell'utilizzare Clang Tidy, in futuro si potrebbe considerare lo sviluppo di un'interfaccia grafica per esso, con la possibilità di scegliere quali fix applicare, semplificandone notevolmente l'utilizzo.

## Ringraziamenti

Il mio primo ringraziamento va ai *miei genitori* che mi hanno permesso di raggiungere questo importante traguardo, ai quali dedico questo lavoro. Ringrazio infinitamente il Professore *Enea Zaffanella*, mio relatore e maestro dentro e fuori l'università: non scorderò mai il discorso sul *bagno d'umiltà*.

Ringrazio i miei amici *Francesco Pietralunga, Gabriele Etta, Tommaso Campari, Renato Garavaglia* e tutti i ragazzi dell' *aula 3*, che hanno dato un senso ad ogni inutile pausa caffè.

Ringrazio i miei fratelli *Amine, Yassine e Bilal* che si sono sempre interessati del mio percorso universitario.

Un ringraziamento va alla professoressa *Angela Superchi*, al professor *Giuseppe Praticò* e alla professoressa *Carla Bacchi Modena*, che mi hanno fatto scoprire il magnifico mondo dell'informatica.

Ringrazio i miei colleghi del *Circolo del Castellazzo* e dell' *F.B. Services Srl*, che mi hanno insegnato tantissimo sotto molti aspetti, in particolare ringrazio *Yassine Riouak* e *Matteo Agnetti*.

*Un ringraziamento speciale ad una persona speciale. Grazie Marta.*

## Elenco delle figure

1	Macro schema di un compilatore. . . . .	2
2	Fasi di compilazione: In blu il <b>front-end</b> ed in rosso il <b>back-end</b>	3
3	Clang e LLVM. . . . .	6
4	Struttura della macro classe Decl, con le principali classi derivate, dell'albero di sintassi astratta di Clang. . . . .	7
5	Dump dell'AST sul file <i>main.cpp</i> . . . . .	9
6	Gestione dei tipi annidati in <i>Clang</i> . . . . .	10
7	Struttura della macro classe Type, con le principali classi derivate, dell'albero di sintassi astratta di Clang. . . . .	11
8	Struttura della macro classe Stmt, con tutte le classi derivate e divisi per tipologia, dell'albero di sintassi astratta di Clang. . . . .	12
9	Esempio di CompoundStmt . . . . .	13
10	Struttura della macro classe Expr, con le principali classi derivate, dell'albero di sintassi astratta di Clang. . . . .	13
11	Albero di sintassi astratta del codice. . . . .	14
12	Una copia di A e una copia di B . . . . .	26
13	Due copie di A e una copia di B . . . . .	26
14	Una copia di A e due copie di B . . . . .	27
15	Una copia di A e una copie di B . . . . .	27
16	Patch del File: AvoidBindCheck.cpp . . . . .	42
17	Patch del File: modernize-avoid-bind.cpp . . . . .	43
18	Link della revisione: <a href="https://reviews.llvm.org/D25649">https://reviews.llvm.org/D25649</a> . . . . .	44

## Riferimenti bibliografici

- [1] **Software Modernization**  
[https://en.wikipedia.org/wiki/Software\\_modernization](https://en.wikipedia.org/wiki/Software_modernization)
- [2] **Compilers: principles, techniques & tools: The Structure of a Compiler**  
V. Aho, S. Lam, R. Sethi, J. Ullman
- [3] **Difference Between Token & Lexeme**  
<http://stackoverflow.com/questions/14954721/what-is-the-difference-between-token-and-lexeme>
- [4] **What is Static Analysis?**  
<http://clang-analyzer.llvm.org/>
- [5] **New LLVM C Front-End 2007**  
<http://llvm.org/devmtg/2007-05/09-Naroff-CFE.pdf>
- [6] **The Clang AST - Manuel Klimek**  
<https://www.youtube.com/watch?v=VqCkCDFLSc&t=436s>
- [7] **Clang-Tidy Modernize**  
<http://clang.llvm.org/extra/clang-tidy/>
- [8] **Programming Languages — C++ [depr.c.headers]**  
<https://isocpp.org/files/papers/N3690.pdf> [depr.c.headers]
- [9] **Move Semantics, Perfect Forwarding and Rvalue references**  
<https://skillsmatter.com/skillscasts/2188-move-semanticsperfect-forwarding-and-rvalue-references>
- [10] **Parma Polyhedra Library**  
<http://bugseng.com/it/prodotti/ppl>
- [11] **[cfe-dev] [Clang-tidy] applying fixes multiple times on same file**  
<http://lists.llvm.org/pipermail/cfe-dev/2015-December/046566.html>