UNIVERSITA' DEGLI STUDI DI PADOVA
FACOLTA' DI INGEGNERIA

Tesi di Laurea

Un'interfaccia C per il software di Programmazione Lineare Intera CBC della Coin-Or

Relatore: Ch.mo Prof. Matteo Fischetti

Laureando: Tommaso Normani

Corso di Laurea in Ingegneria Informatica

Anno Accademico 2007 – 2008

Introduzione

Scopo dell'elaborato

L'obiettivo iniziale di questo elaborato è stato quello di costruire un'interfaccia tra la distribuzione CBC ed un altro linguaggio di programmazione, nel nostro caso il C, in maniera tale da poter sfruttare le potenzialità di questo prodotto software open-source in altri contesti; come secondo fine, ma ugualmente importante, si è deciso di redigere una guida, basata sulle mia esperienza al riguardo, a beneficio di chi volesse far uso del software COIN OR.
Questo documento ha come scopo primario quello di far conoscere ai potenziali utilizzatori le problematiche principali che si potrebbero presentare in fase di sviluppo, permettendo loro allo stesso tempo di impratichirsi nel minor tempo possibile.

Perché usare il software COIN-OR?

Per prima cosa presentiamo il "progetto": la **CO**mputational **IN**frastructure for **O**perations **R**esearch (COIN-OR) è un'iniziativa nata per spronare lo sviluppo di software open-source mirato alla Ricerca Operativa.
Ci si può chiedere, perché open-source? Bè, rispondere a questa domanda è piuttosto semplice: è sufficiente le persone leggano, utilizzino e di conseguenza correggano il codice, per ottenere un aggiornamento ed un miglioramento del software. Questo ha portato ad un importante risultato, infatti ora il software open source offre ottime performance ed è altamente affidabile e proprio su questo codice sono stati sviluppati molti altri programmi.
Inoltre, quanto detto poc'anzi è una maniera per consentire alle persone di confrontarsi, mettere in campo le proprie idee e ottenere nuovi risultati, che portano a continui miglioramenti.

I progetti COIN-OR

Di tutto quello che la COIN (comunemente abbreviata) mette a disposizione, nel seguito di questa guida troverete solo una piccola parte, quella riguardante la cosidetta MIP (Mixed Integer Programmation). In linea di massima alcune sezioni di questa pubblicazione si possono adattare a qualunque progetto COIN, mentre altre ovviamente sono dedicate.
Inoltre, è possibile effettuare il download di ogni distribuzione in più versioni dal rispettivo sito internet, ognuna adattata a diverse esigenze.
Infatti sono disponibili versione precompilate, contenenti i file binari in archivio tar da scompattare, adatte per chi deve utilizzare il software per cose semplici quali ad esempio un problema di Programmazione Lineare Intera (PLI d'ora in poi); oltre a queste, ci sono le distribuzioni per sviluppatori, che forniscono sorgenti, librerie, makefile, doxydoc e quant'altro occorre per poterci lavorare sopra.
Andiamo ora a presentare il vero punto focale di questo documento: la distribuzione CBC e ovviamente i progetti correlati.

Distribuzione CBC

Introduzione

CBC, non è in grado di operare da solo, ma fa affidamento su altre componenti del repertorio COIN. Infatti, per funzionare ha bisogno di un risolutore di Linear Programming (**C**OIN **L**inear **P**rogramming) e di un'interfaccia che gli permetta di comunicare con l'utente per ciò che concerne il risolutore, il COIN **O**pen **S**olver **I**nteface. In parole povere, qualsiasi risolutore LP che disponga di un'interfaccia OSI può essere utilizzato con CBC e non necessariamente il Clp, garantendo a CBC stesso una certa portabilità. In ogni caso ci si aspetta che Clp sia il risolutore più comunemente usato in abbinamento, dato che è quello nativo COIN-OR.
Tecnicamente parlando CBC accede al risolutore attraverso la OSI, quindi d'ora in poi ci farà comodo "confondere" l'interfaccia con il risolutore vedendoli come una black box inscindibile.
Per quel che riguarda il generatore di tagli CBC si basa sul COIN **C**ut **G**eneration **L**ibrary, ma in maniera del tutto analoga al risolutore LP, qualsiasi generatore di tagli che sia scritto secondo gli standard del Cgl può essere utilizzato. Comunque, c'è da dire che in questo caso la portabilità è minore (o quantomeno richiede un maggiore lavoro) dato che alcuni dei tagli presenti nel Cgl fanno riferimento ad altre parti dei progetti COIN, come ad esempio il generatore di tagli di Gomory (implementato nel Cgl) fa riferimento alla funzionalità di fattorizzazione presente nella classe CoinFactorization.

CBC

Analizzeremo ora per sommi capi come funziona l'algoritmo implementato, facendo qualche riferimento alle principali classi utilizzate, date queste assunzioni:

- Se non diversamente specificato, il problema che deve essere ottimizzato è un problema di minimizzazione;
- Modello e problema verranno usati come sinonimi;
- Useremo la convenzione di aggiungere un underscore alle variabili per poter distinguere i dati membri di una certa classe.

L'algoritmo è:

- Passo 1. (Bound)

Dato un modello di tipo MIP(Mixed Integer Programming) da minimizzare, ci si occupa prima di tutto dei requisiti di integrità. In secondo luogo, viene risolto il modello lineare appena ricavato per ottenere un bound inferiore al valore della funzione obiettivo del modello. Se la soluzione ottima risultante dall'LP (cioè il limite inferiore ottenuto poc'anzi) ha valore intero per le variabili intere del problema MIP, siamo arrivati. Inoltre sapendo che qualsiasi soluzione MIP-feasible fornisce un limite superiore alla funzione obiettivo, allora se limite inferiore e superiore coincidono la soluzione è ottima.

- Passo 2. (Branch)

Se non avviene quanto detto al Passo 1 vuol dire che esistono una o più variabili con valore non intero. Basta quindi scegliere una di queste(poniamo ad esempio sia 1.3) e fare

il branch(vedi classi A e B): vengono creati 2 nodi, uno con la variabile di branching con un limite superiore di 1.0 e l'altro avente un limite inferiore di 2.0. Questi 2 nodi vengono aggiunti all'albero di ricerca. Ora si procede con un ciclo:

While(l'albero di ricerca non è vuoto) {

- Passo 3. (Choose Node)

Si estrae il nodo da elaborare dall'albero (vedi classi C e D).
Passo 4. (Re-optimize LP)
Viene creato un rilassamento LP e e lo si risolve.

- Passo 5. (Bound)

Si guarda la soluzione ottima che si è ottenuta al Passo 4 e si cerca di potare il nodo in base ad uno dei seguenti criteri:

  - se è infeasible, pota il nodo.
  - se il valore della soluzione ottima dell'LP del nodo eccede il limite superiore corrente, pota il nodo.
  - se la soluzione LP ottima del nodo non eccede il limite superiore corrente ed è feasible, allora aggiorna il limite superiore, la miglior soluzione MIP conosciuta e pota il nodo per ottimalità.

- Passo 6. (Branch)

Se non ci era possibile potare il nodo, allora branch nuovamente. Ripetiamo quindi il Passo 2.

}

Questo è in linea di massima il funzionamento. Se nella fase di ottimizzazione usiamo dei tagli per stringere il rilassamento LP (vedi classi E e F), allora otteniamo l'algoritmo branch and cut.

Classi utilizzate nella descrizione dell'algoritmo:

	Nome Classe/i	Breve descrizione
(A)	CbcBranch...	Definiscono la natura della discontinuità MIP, La più semplice è una variabile che deve prendere un valore intero.
(B)	CbcNode	Viene stabilita quale variabile/entità deve essere "branchata" al passo successivo. Gli utenti avanzati probabilmente interagiranno unicamente con questa classe per settare i parametri di CbcModel.
(C)	CbcTree	Tutti i modelli non risolti possono essere pensati come essere nodi su un albero (questa classe), dove ciascun nodo(modello) può ramificare a sua volta due o più volte.
(D)	CbcCompare...	Vengono utilizzate nella determinazione di quale, fra i nodi ancora inesplorati dell'albero, deve essere considerato al prossimo passo.
(E)	CglCutGenerators	Come già detto andrebbe bene qualunque generatore di tagli costruito in maniera analoga al CGL, ma dato che questo è quello COIN è allo stesso tempo quello di riferimento. I generatori di tagli vengono passati a CBC parametrizzati in maniera tale da poter essere modificati ad ogni tentativo. Per un dato problema, bisognerebbe provare tutti i generatori di tagli per determinare qual'è veramente effettivo.
(F)	CbcHeuristics	Le euristiche sono veramente importanti per ottenere una soluzione valida rapidamente; alcune euristiche sono già disponibili, ma qui vale la pena scriverne di specializzate per il modello che si sta analizzando.

La classe di maggior rilievo in CBC è CbcModel, infatti quest'ultima è dove viene fatta la maggior parte dell'impostazione dei parametri. L'approccio più minimalista alla classe CbcModel prevede unicamente due metodi, CbcModel(OsiSolverInterface & linearSolver) come costruttore, e branchAndBound() per risolvere il problema.
Diamo quindi un'occhiata ad un esempio abbastanza semplice preso dalla documentazione e cerchiamo di capirne il funzionamento.

Esempio di Branch-and-Bound (minimum.cpp)

Questo programma mostra come effettuare un semplice Branch and Bound con CBC.
(Il codice è tratto dalla classe minimum.cpp della distribuzione CBC della COIN-OR ®)

```
// Copyright (C) 2005, International Business Machines
// Corporation and others.  All Rights Reserved.

#include "CbcModel.hpp"

// Using CLP as the solver
#include "OsiClpSolverInterface.hpp"

int main (int argc, const char *argv[])
{
  OsiClpSolverInterface solver1;

  // Read in example model in MPS file format
  // and assert that it is a clean model
  int numMpsReadErrors = solver1.readMps("../../Mps/Sample/p0033.mps","");
  assert(numMpsReadErrors==0);

  // Pass the solver with the problem to be solved to CbcModel
  CbcModel model(solver1);

  // Do complete search
  model.branchAndBound();

  /* Print the solution.  CbcModel clones the solver so we
     need to get current copy from the CbcModel */
  int numberColumns = model.solver()->getNumCols();

  const double * solution = model.bestSolution();

  for (int iColumn=0;iColumn<numberColumns;iColumn++) {
    double value=solution[iColumn];
    if (fabs(value)>1.0e-7&&model.solver()->isInteger(iColumn))
      printf("%d has value %g\n",iColumn,value);
   }
```

```
  return 0;
}
```

Il codice nell'esempio qui sopra, crea un oggetto della classe OsiClpSolverInterface e attraverso questo legge un file mps, contenente il problema da elaborare. Se la lettura avviene priva di errori, il programma passa il contenuto a CbcModel che lo risolve mediante l'algoritmo di Branch and Bound. La parte del programma che risolve il modello è piuttosto semplice, si riduce unicamente alla chiamata dell'algoritmo, ma prima bisogna crearlo e popolarlo con i dati presenti nel file. Il programma termina con una stampa a video.
La parte senza alcun dubbio di maggior interesse, è l'interazione fra OSI e CBC di cui ci occuperemo ora e che era stata già accennata prima.

Relazione tra OSI e CBC

Osserviamo con attenzione il codice nella parte di "inizializzazione": il costruttore di CbcModel prende un puntatore ad un'oggetto di tipo OsiSolverInterface, ne crea un clone e d'ora in poi userà questa nuova istanza del risolutore (ricordiamo che noi usiamo le parole risolutore e OsiSolverInterface indifferentemente). Le due istanze sono e restano **NON** sincronizzate fino a quando non è l'utente stesso a decidere di farlo manualmente. Come fare ciò?
Bisogna utilizzare un comando del tipo:

```
  solver1 = model.solver();
```

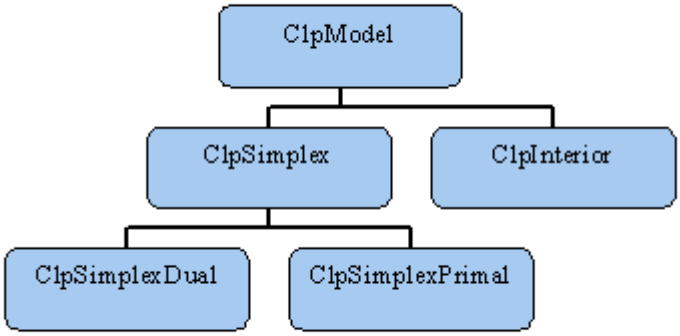
in cui il metodo solver() di CbcModel restituisce un puntatore ad un oggetto di tipo OsiSolverInterface e lo assegna al risolutore clonato.
Per convenienza, molti dei metodi OSI per accedere ai dati del problema hanno nomi identici in CbcModel(sfruttando il polimorfismo di C++). Infatti è più facile scrivere model.getNumCols() che model.solver()->getNumCols().
CbcModel aggiorna il suo risolutore automaticamente in determinati punti focali durante l'algoritmo; in questi punti, l'informazione contenuta nell'istanza di tipo CbcModel coinciderà con quella nel suo clone. In altri momenti dell'esecuzione, come già detto, le due copie possono differire per le informazioni contenute.
Mentre tutti i metodi OSI usati in minimum.cpp hanno il loro equivalente in CbcModel, ci sono alcuni metodi OSI che non lo hanno. Ad esempio, se il programma ha prodotto una quantità eccessiva di output indesiderato, esiste un metodo OSI per ovviare a ciò come si vede nella seguente riga di codice:

```
model.solver()->setHintParam(OsiDoReducePrint,true,OsiHintTry);
```

di questo, non esiste l'equivalente metodo in CbcModel.
Quindi, nel complesso OsiSolverInterface e CbcModel funzionano in maniera totalmente analoga, salvo alcune piccole differenze come è stato evidenziato sopra.

CLP

Spendiamo ora qualche parola sul CLP, dato che questo è una colonna portante del CBC. La gerarchia delle classi nella sua versione più elementare è semplice. Suddiviso in tre livelli (raffigurati qui sotto), i primi 2 (partendo dall'alto) contengono tutti i dati del problema che defiscono il problema, mentre l'ultimo contiene principalmente gli aspetti algoritmici del CLP.



La maniera secondo lo standard OSI, per controllare l'ottimalità è chiamare model.isProvenOptimal().
Sono inoltre disponibili:

- isProvenPrimalInfeasible();
- isProvenDualInfeasible();
- isPrimalObjectiveLimitReached();
- isDualObjectiveLimitReached();
- isIterationLimitReached();
- isAbandoned();

quest'ultimo nel caso si abbandoni il problema perchè sono stati riscontrati problemi numerici.
Le due strategie possibili, cioè OSI e OSL, differiscono principalmente nel modo di ritornare i dati; nel primo caso array in forma costante, nel secondo in forma non costante. Normalmente risulta più facile lavorare con il secondo tipo, da qui molti algoritmi vengono sviluppati in CLP-style (cioè l'OSL).

Scopo	OSI-style	OSL-style
Primal column solution	const double * getColSolution()	double * primalColumnSolution()
Dual column solution	const double * getRowPrice()	double * dualColumnSolution()
Primal row solution	const double * getRowActivity()	double * primalRowSolution()
Dual column solution	const double * getReducedCost()	double * dualColumnSolution()
Number of rows in model	int getNumRows()	int numberRows()
Number of column in model	int getNumCols()	int numberColumns()

Coin-MP

Presentazione

CoinMP è una libreria dinamica linkata (DLL), che supporta la maggior parte delle funzionalità di CLP, CBC e CGL. Questa distribuzione di CoinMP.dll della COIN-OR ha come vantaggi di essere altamente portabile, non richiedere alcuna compilazione, è facilmente utilizzabile e può essere chiamata da qualsiasi altra applicazione Windows. Vedremo in seguito qualche piccolo riferimento al riguardo.

Descrizione

Prima di continuare, facciamo un breve sunto di cosa questa DLL mette a disposizione in maniera simile ad un doxygen.

Tipo ritornato	Nome metodo	Descrizione
char*	CoinGetSolverName(void)	Restituisce il nome del risolutore in uso
char*	CoinGetVersionStr(void)	Restituisce la versione del risolutore in uso
HPROB	CoinCreateProblem(char *ProblemName)	Crea un'istanza di tipo HPROB del problema avente nome passato come parametro
int	CoinReadFile(HPROB hProb, int FileType, char *ReadFilename);	Legge il file il cui nome è passato come parametro, del tipo passato come parametro (il tipo di file che si aspetta, vedi costanti a fine file coinmp.h) e lo carica nell'istanza del problema creata precedentemente
char*	CoinGetSolutionText(HPROB hProb, int SolutionStatus)	Restituisce la soluzione del problema
int	CoinGetObjectValue(HPROB hProb)	Restituisce il valore della funzione obiettivo associata al problema che si sta analizzando
int	CoinGetMipBestBound(HPROB hProb)	Restituisce il migliore limite per il modello mip associato al problema che si sta analizzando

Come si può facilmente osservare, i metodi, sono piuttosto semplici e in quantità decisamente minore di quella fornita dalla libreria CBC. Ovviamente questi non sono tutti quelli messi a disposizione, ma nonostante questo non si ha certo il grado di "personalizzazione" che si può ottenere utilizzando la distribuzione CBC appieno.
Questa DLL consente a tutti gli effetti di risolvere un problema di programmazione lineare intera mediante branch and cut, ma come questa arrivi al risultato deve essere preso a

scatola chiusa (come peraltro noi supporremo quando faremo l'interfaccia per il C, che anche CoinMP stesso fornisce).
Questa ipotesi un po' restrittiva fa perdere parte delle potenzialità del prodotto, ma è obbligatoria in quanto il C è differente e meno versatile del C++; proprio per questo ci sono cose che in C++ saranno possibili, mentre in C no e di queste dovremo fare a meno.

Guida all'uso del CBC

Introduzione

Fatta questa presentazione che permette di avere un'idea più chiara di cosa stiamo andando ad utilizzare, passiamo alla guida, che si basa sulla mia esperienza nell'uso del software CBC e progetti correlati. Consiglio vivamente di leggere con cura **tutto** ciò che segue e di trarre le proprie conclusioni solo in ultima analisi, perché come vedremo non vi è una sola maniera per fare le cose.

Scelta dell'ambiente di lavoro

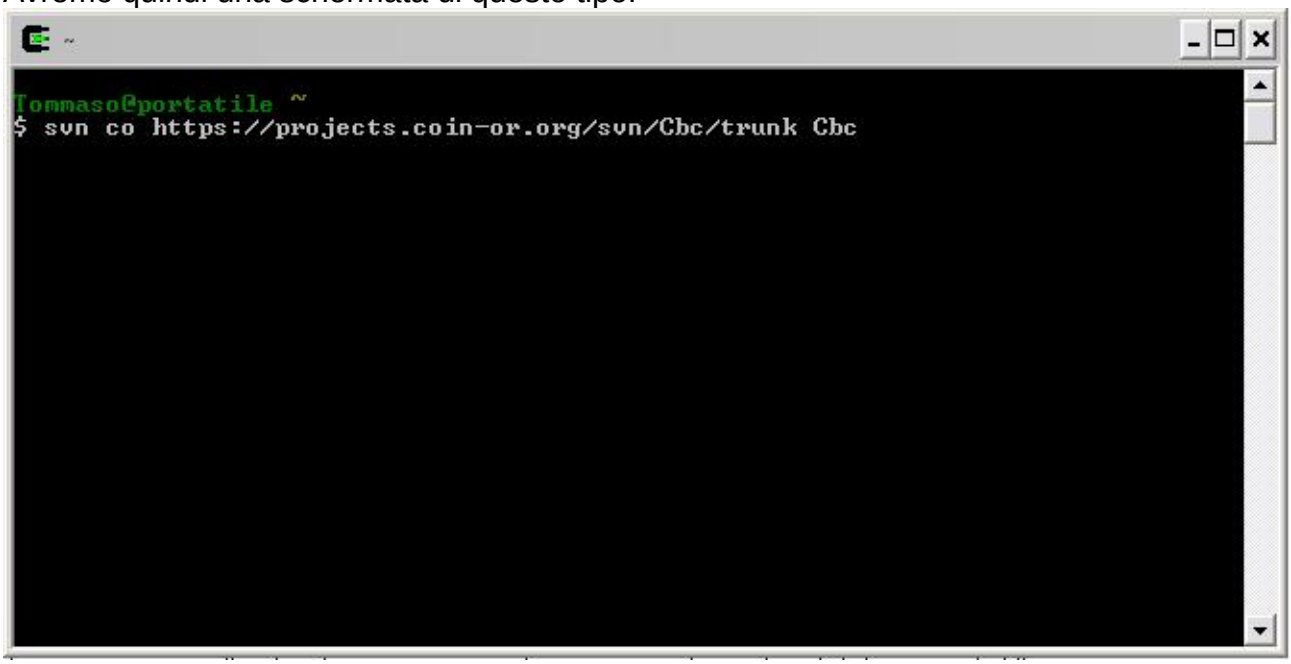
Primo passo, senza alcun dubbio, stabilire il sistema operativo con il quale si intende lavorare se Unix/Linux o Windows(strettamente Windows oppure Cygwin).
La cosa apparentemente banale, non lo è affatto, specie nel caso (come il mio) in cui si decida di utilizzare usare Cygwin sotto piattaforma Windows.
Si è optato per questa scelta per avere una visione differente delle cose, infatti normalmente questo software viene utilizzato in ambiente Unix/Linux; si è avuto così modo di testare il comportamento del prodotto COIN se usato mediante un emulatore in ambiente Windows, cosa che ha causato qualche grattacapo, successivamente risolto come spiegato nel seguito della relazione.

Ottenere e installare il software

Innanzitutto occupiamoci di ottenere una versione completa della distribuzione CBC stabile, che all'atto della redazione di questa guida è la 2.0, oppure una in corso di sviluppo. So che questa parte sembra semplice, ma ho trovato numerosi (per non dire la maggior parte) dei problemi proprio in questa fase.
Nel caso non si disponga di Cygwin, o non si sia intenzionati ad usarlo, si può effettuare il download sempre mediante svn utilizzando Tortoise(http://tortoisesvn.tigris.org/). La spiegazione al riguardo è successiva a quella per Cygwin.
A questo punto, usando la nostra shell opportunamente configurata(di default non installa i pacchetti per make, configure e svn pertanto sarà vostro compito in fase di installazione assicurarsi siano presenti) ci posizioniamo nella directory in cui vogliamo creare la nostra distribuzione di Cbc e lanciamo il comando di svn, come segue:

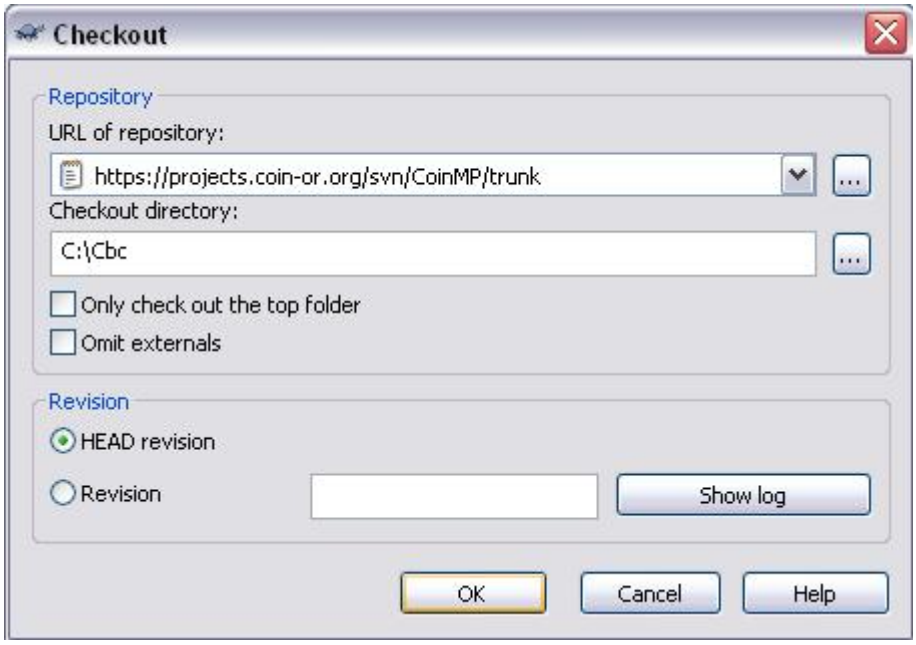
- per ottenere la versione dal Trunk, che è quella in corso di sviluppo
  svn co https://projects.coin-or.org/svn/Cbc/trunk Cbc
- per l'ultima versione stabile invece
  svn co https://projects.coin-or.org/svn/Cbc/stable/2.0 Cbc

Questa versione è quella che ora si può reperire anche alla pagina del progetto(fino a poco tempo fa era da aggiornare, faceva riferimento alla 1.1).
Avremo quindi una schermata di questo tipo:



A questo punto basta accettare permanentemente (p) il certificato che ci verrà proposto e attendere la fine del download.
Per quel che riguarda Tortoise svn, gli indirizzi sono gli stessi basta quindi seguire i passaggi riportati negli screen che seguono:

e in maniera del tutto analoga a Cygwin, solo per via grafica:



Utilizzando svn mediante Tortoise, il programma eseguirà un check di genuinità in ogni file ottenuto (identificabile attraverso un punto esclamativo verde). In seguito a quello che stiamo per fare perderemo tale check dato che verranno alterate le dimensioni di alcuni file. La cosa non deve creare alcuna preoccupazione dato che al primo aggiornamento mediante svn verrà ripristinato.

Inoltre, se il download dovesse fallire per un qualunque motivo, ma in particolare perché viene individuato qualche file corrotto, basta utilizzare cleanup(in Cygwin) oppure l'apposito cleanup presente nel menù di Tortoise(sempre col tasto destro del mouse) e cancellare manualmente il file corrotto. Fatto questo, rieseguire l'svn.

Terminata questa fase si arriva ad una prima scelta, se si prova a compilare la distribuzione in ambiente Cygwin e successivamente in ambiente VC++(Visual Studio C++, nella mia esperienza la versione 9) il secondo genera migliaia di errori in fase di compilazione. Le cose apparentemente non dovrebbero essere legate, dato che Cygwin fa riferimento ai file config, make ed install mentre VC++ a dei file progetto che si trovano in un'apposita cartella, ma nonostante questo compilare in entrambi gli ambienti non sembra essere possibile.
Quindi le 2 situazioni verranno analizzate separatamente:

- Cygwin

Vediamo prima di tutto il blocco di istruzioni da eseguire da shell:

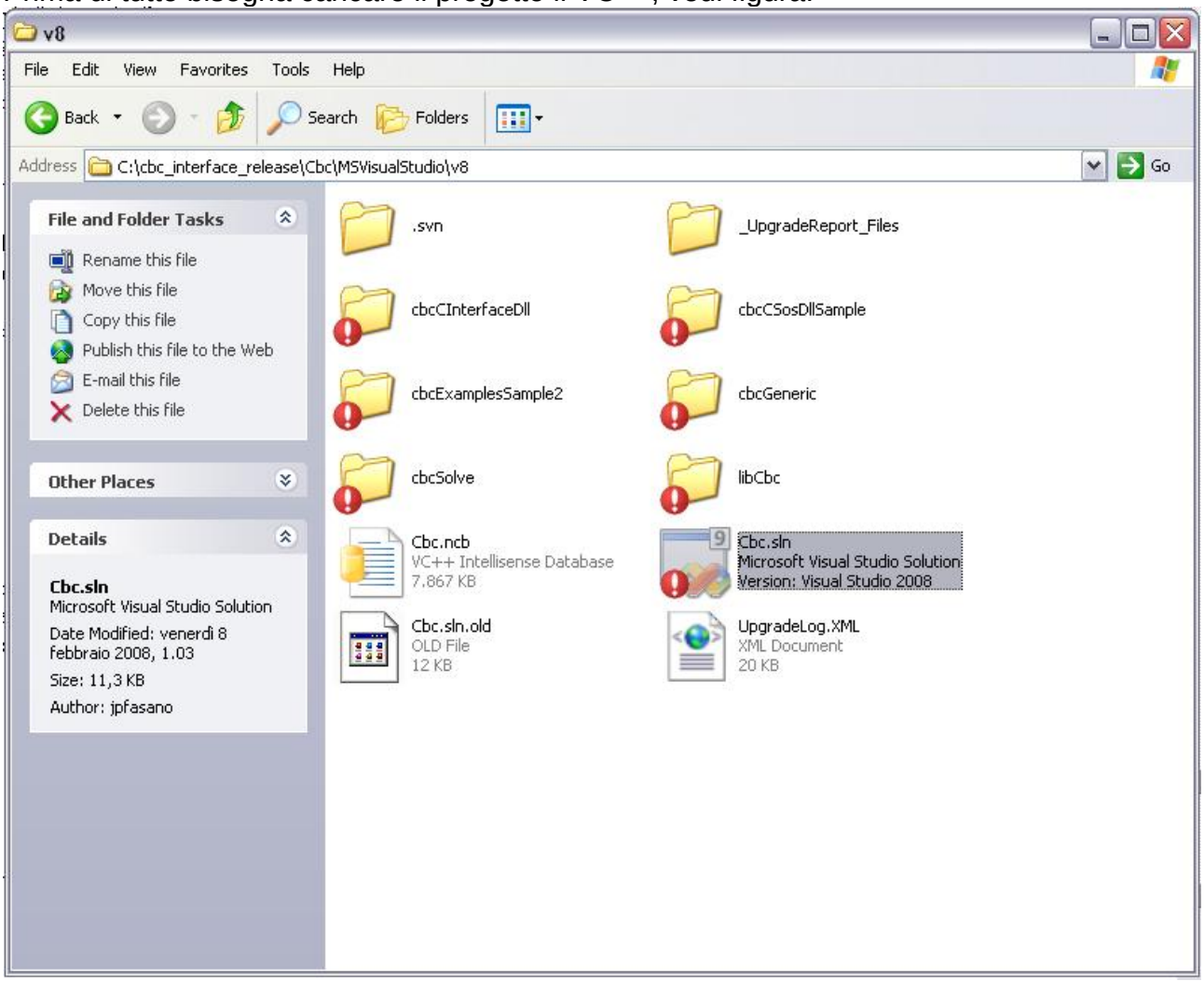
```
cd <nome cartella della distribuzione>
./configure [-c]
make
make test
make install
```

il –c è opzionale in quanto riguarda l'utilizzo o meno della cache. Fare uso della cache in fase di configurazione significa utilizzare informazioni ottenute da configurazioni precedenti riguardanti questa distribuzione.

In seguito vedremo cosa comporta l'esecuzione di quei tre comandi di make, in particolare la creazione di un makefile adeguato che consenta di ottenere file anche se collocati in cartelle diverse e di impostare il compilatore secondo quanto serve.
Questo blocco di operazioni richiede una certa quantità di ram (>512mb) per poter essere eseguita correttamente, altrimenti porta a crash del sistema Cygwin per errori nelle chiamate ricorsive (in quanto trova le risorse occupate non avendo più spazio da allocare). La cosa risulta essere più grave di quel che si può pensare, perché nel tentativo di allocare spazio inizia a sovrascrivere aree di memoria fino ad incontrare un processo di sistema, causando il crash (di Windows).
Per ovviare a questo, nel caso in cui vi ritroviate una quantità di ram non adeguata, potete andare in modalità provvisoria di Windows e da lì eseguire con successo lo stesso set di comandi.

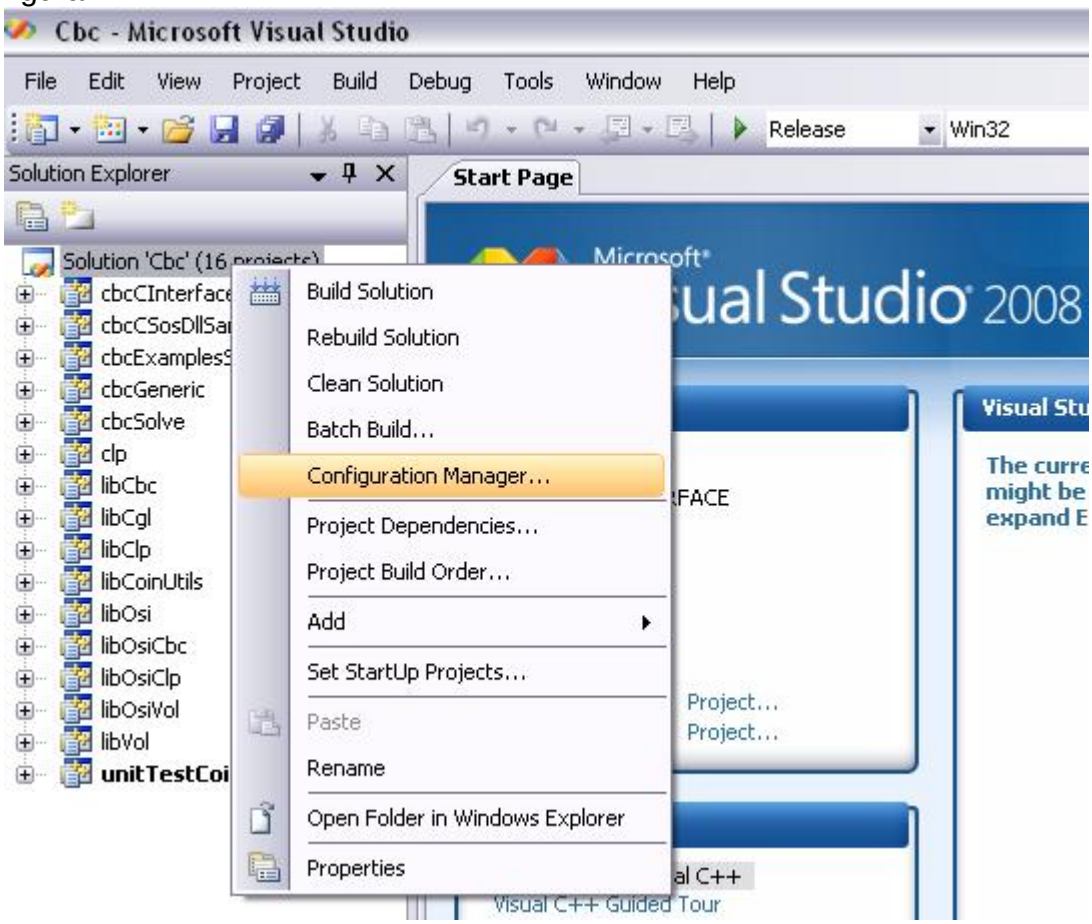
- VC++

Veniamo ora a Visual C++
Prima di tutto bisogna caricare il progetto il VC++, vedi figura:

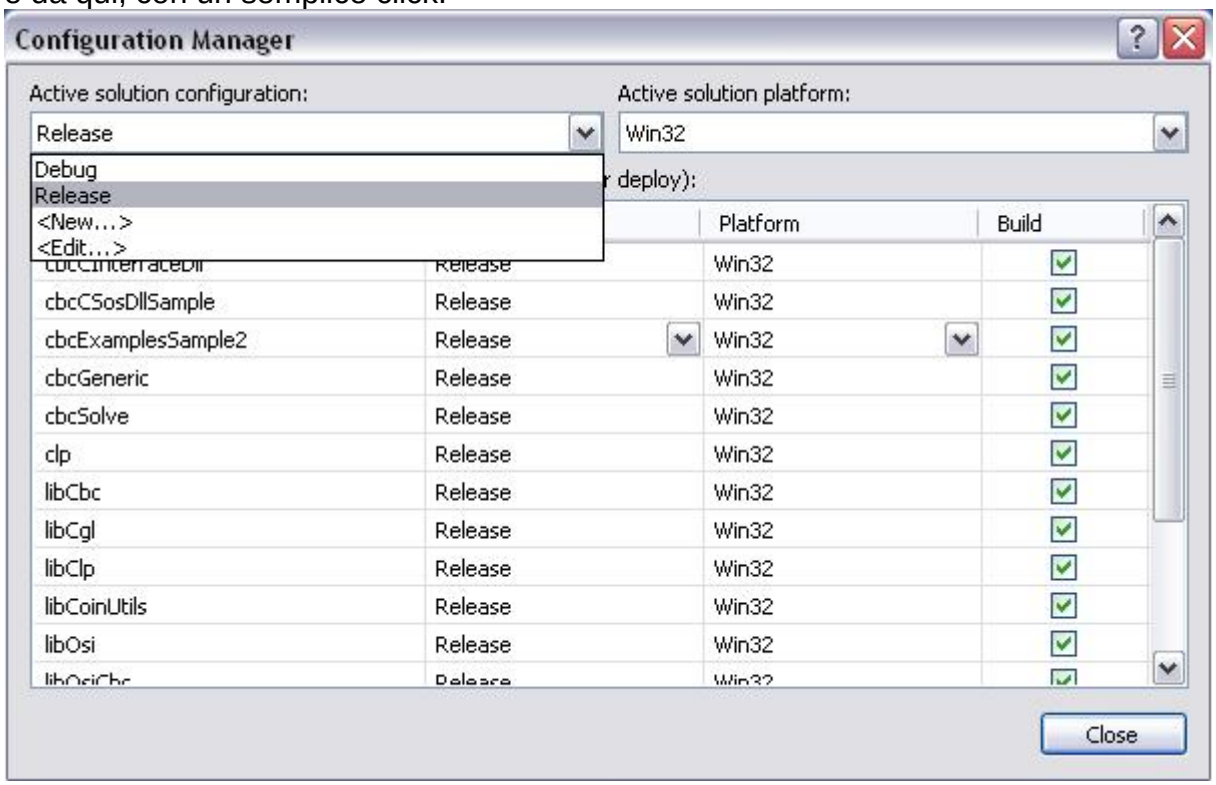


Il file in questione verrà aggiornato in avvio alla versione di VC++ in vostro possesso(se superiore alla 8).
Nel caso sia una versione inferiore alla 9, al percorso .\Cbc\MSVisualStudio\ troverete le distribuzioni per versione 6, 7 e 8. In ogni caso, come si può facilmente intuire, bisogna quantomeno il vostro VC++ sia versione 6.

Esattamente come in Cygwin, anche VC++ è in grado di gestire cartelle e impostazioni di compilazione, in quanto il file Cbc.sln contiene tutte le informazioni necessarie per l'intera distribuzione. Nella fattispecie, l'unica cosa richiesta all'utente per la compilazione del progetto è passare dalla modalità "debug" a quella "release", seguendo quanto riportato in figura:



e da qui, con un semplice click:



e a questo punto basta compilare, gli output verranno forniti nella cartella della soluzione (quella dove abbiamo reperito il file Cbc.sln), oppure in qualche altra destinazione se avete fornito specifiche diverse.

Quanto appena detto, non è ovviamente valido per altri progetti COIN in quanto non è detto che sia fornita un'interfaccia VC++, oppure che la sequenza di comandi nella shell Cygwin sia la stessa. In linea di massima, questo può dare un'idea in quanto sarà qualcosa di simile anche se non uguale.

Interfacciare il CBC

Un esempio pratico:CBC e il C

Arrivati a questo punto è finalmente possibile utilizzare la nostra distribuzione. Veniamo quindi allo scopo del mio elaborato, creare un'interfaccia al C, una struttura in grado di rendere l'intero CBC una black box funzionante solo con i metodi che si decide di implementare.
Al solito le scelte per questo wrapper sono molteplici, ma si è scelto di procedere come segue perché è la soluzione che garantisce più elasticità e semplicità.
Bisogna, per prima cosa fare chiarezza sui due problemi principali che si presentano: uso dell'extern e problemi di compilazione (mangling, ovvero la decorazione dei nomi in C++).

Uso dell'extern

Per comprendere meglio il concetto e' necessario sapere che in C e in C++ l'unita' di compilazione e' il file; un programma puo' consistere di piu' file che vengono compilati separatamente e poi linkati (collegati) per ottenere un file eseguibile. Facciamo un esempio:

```
// File a.cpp
  int a = 5;

// File b.cpp
  extern int a;

  int GetVar() {
    return a;
  }
```

Il primo file dichiara una variabile intera e la inizializza, il secondo (trascuriamone per ora la prima riga di codice) dichiara una funzione che ne restituisce il valore. La compilazione del primo file non e' un problema, ma nel secondo file GetVar() deve utilizzare un nome

dichiarato in un altro file; perche' la cosa sia possibile bisogna informare il compilatore che tale nome e' dichiarato da qualche altra parte e che il riferimento a tale nome non puo' essere risolto se non quando tutti i file sono stati compilati; solo il linker quindi puo' risolvere il problema collegando insieme i due file.
Il compilatore deve dunque essere informato dell'esistenza della variabile al fine di non generare un messaggio di errore; tale operazione viene effettuata tramite la keyword extern.
In effetti la riga extern int a; non dichiara un nuovo identificatore, ma dice "La variabile intera a e' dichiarata da qualche altra parte, lascia solo lo spazio per risolvere il riferimento".
Naturalmente extern si puo' usare anche con le funzioni (anche se come vedremo e' ridondante):

```
// File a.cpp
  int a = 5;

  int f(int c) {
    return a+c;
  }


// File b.cpp
  extern int f(int);

  int GetVar() {
    return f(5);
  }
```

Si noti che e' necessario che extern sia seguita dal prototipo completo della funzione.
Dopo aver fatto un po' di chiarezza riguardo l'uso di extern, veniamo a ciò che realmente occorre a noi. Partendo dal fatto che il C++ ha un'alta compatibilita' col C, è facile renderci conto che e' possibile interfacciare codice C++ con codice C; anche in questo caso l'aiuto ci viene dalla keyword extern. Per poter linkare un modulo C con un modulo C++ e' necessario indicare al compilatore le nostre intenzioni:

```
// Contenuto file C++
  extern "C" int CFunc(char*);
  extern "C" char* CFunc2(int);

// oppure per risparmiare tempo
  extern "C" {
    void CFunc1(void);
    int* CFunc2(int, char);
  }
```

Osserviamo un secondo la dichiarazione: la keyword extern svolge il ruolo detto poc'anzi, mentre la presenza di "C" serve a indicare che bisogna adottare le convenzioni del C sulla codifica dei nomi. Ma perché questo? Perché in fase di compilazione, C e C++ hanno un

modo di codificare internamente i nomi molto differente, in particolare C++ aggiunge un sacco di informazioni che il C non ha.

Mangling (o decorazione dei nomi)

Come già detto precedentemente, il C++ ha un modo particolare per codificare gli identificatori, siano essi di variabili, metodi o quant'altro. Vediamo nel dettaglio questo cosa comporta.
Nel campo della compilazione il name mangling (più propriamente chiamato decorazione dei nomi)è una tecnica usata per risolvere problemi causati dalla necessità di risolvere nomi univoci per le entità nei linguaggi di programmazione moderni.
Questo meccanismo fornisce una maniera per codificare le informazioni aggiuntive riguardo al nome della funzione, la struttura, la classe o altri tipi di dato in maniera da passare più informazioni semantiche possibili dal compilatore al linker.
Il bisogno di tutto ciò, cresce ove il linguaggio consente a entità differenti di avere identificatori con il medesimo nome nonostante questi occupino un differente namespace (dove per namespace si intende un modulo, una classe oppure esplicitamente una direttiva di namespace).
Qualunque codice oggetto prodotto dal compilatore viene di solito collegato con altri segmenti di codice oggetto (prodotti dallo stesso o da un altro compilatore compatibile) attraverso un programma chiamato linker. Quest'ultimo, per svolgere il proprio lavoro, richiede un grande ammontare di informazioni per ogni entità del programma, come ad esempio il nome di ogni funzione, il numero di parametri e i tipi e così via.
Vediamo ora come si comportano i singoli linguaggi.

- C name decoration in Microsoft Windows

Nonostante in genere linguaggi che non supportano l'overloading degli operatori/funzioni come il C e il Pascal, non richiedano il name mangling, a volte viene utilizzato ugualmente per fornire informazioni aggiuntive riguardo una qualche funzione. Per esempio, i compilatori in ambiente Microsoft Windows adottano una certa quantità di convenzioni sulle chiamate, che determina la maniera in cui i parametri vengono spediti alle subroutines e i risultati ritornati. Dato che convenzioni di chiamata differenti non sono compatibili le une con le altre, i compilatori gestiscono i simboli con codici che dettagliano quale convenzione deve essere utilizzata.
Lo schema di gestione è stato stabilito dalla Microsoft ed è stato seguito da altri compilatori quali Borland and GNU gcc. Inoltre, questo schema si adatta ad altri linguaggi quali Pascal , D, Delphi, Fortran e C#. Questo consente alle subroutines scritte in questi linguaggi di chiamare o essere chiamate da librerie di Windows usando una convenzione diversa da quella di default.
Vediamo cosa succede compilando il seguente esempio C:

```
int _cdecl    f (int x) { return 0; }
int _stdcall  g (int y) { return 0; }
int _fastcall h (int z) { return 0; }
```

_cdecl è il default per le chiamate C, se nessun'altra convenzione di chiamata è specificata esplicitamente.

Un compilatore a 32 bit fornirà il seguente output:

_f
_g@4
@h@4

Questo perchè nello schema di mangling di stdcall e di fastcall le funzioni vengono codificate come _name@X e @name@X, rispettivamente, dove X è il numero di byte in decimale dei parametri della funzione.
Altre convenzioni coinvolgono l'uso di prefissi o l'abbondanza di sottolineature, come ad esempio __func__.

- Name mangling in C++

I compilatori C++ sono quelli che fanno un uso molto spinto del name mangling, ma allo stesso tempo sono i meno standard, dato che ognuno utilizza una sua convenzione.
I primi compilatori C++ furono implementati come traduttori a codice C, che veniva poi compilato da quest'ultimo per produrre codice oggetto; per questo motivo, i nomi simbolici dovevano rispettare le convenzioni del C. Più avanti negli anni, con la nascita di compilatori in grado di produrre direttamente codice macchina o assembly, il linker non era ancora in grado di gestire i simboli C++ e venne in soccorso il name mangling.
A differenza di quanto detto per il C però, il linguaggio C++ non definisce una standard per lo schema di decorazione, così ogni compilatore ha il suo. Combinato col fatto che la decorazione dei nomi in C++ è andata complicandosi (salvare informazioni riguardo classi, template, namespace, overloading di operatori, etc), si giunge facilmente alla conclusione che codice compilato da diversi compilatori non è linkabile, salvo casi sporadici e fortunosi.
Ad esempio, consideriamo le due seguenti definizioni di f() in un programma C++:

```
int f (void) { return 1; }
int f (int)  { return 0; }
void g (void) { int i = f(), j = f(0); }
```

Queste sono funzioni distinte, con nessuna relazione fra loro se non il nome. Se fossero state semplicemente tradotte in C, ci sarebbe stato un errore ovviamente. Mentre il compilatore darà un output di questo tipo:

```
int __f_v (void) { return 1; }
int __f_i (int)  { return 0; }
void __g_v (void) { int i = __f_v(), j = __f_i(0); }
```

Notare come, nonostante non sia necessario, vengano applicate le convenzioni pure a g().

Preprocessing

Vedremo poi guardando il codice, che il nostro lavoro si ridurrà a "poche" righe un po' elaborate e in questo ci verrà in aiuto il preprocessor. Infatti nelle nostre definizioni di metodi faremo uso di identificativi costanti e sarà poi compito di quest'ultimo sistemare le parti complesse.
Ma cos'è il preprocessor? Bè C, ma anche C++, ha un programma speciale al suo interno chiamato preprocessor che permette di definire e associare nomi simbolici a costanti. Questo programma viene eseguito prima del compilatore e durante questa fase(chiamata

per l'appunto di preprocessing) il preprocessor usa la terminologia di macro definita per sostituire ogni nome macro con il suo body, questa operazione viene detta di espansione.
Si possono mettere le definizioni delle macro in qualsiasi punto del programma, l'unica regola da rispettare è che devono essere materialmente scritte prima di dove viene usato il nome associato.
Inoltre, il preprocessor fornisce la possibilità di includere altri file sorgente. Ad esempio, potremmo voler includere alcune librerie C usando la direttiva #include.
Per quel che riguarda le costanti, il corpo di una macro specificata da una #define può essere una qualsiasi stringa di caratteri o numerica. Ad esempio:

#define TRY "prova"

Questo vuol dire, che durante il preprocessing ogni volta che verrà trovato il nome macro TRY, verrà sostituito con "prova".

Scriviamo l'interfaccia

Quanto è stato detto sopra è ciò che rende difficile scrivere quest'interfaccia, anche se come vedremo facendo le cose un passo per volta le difficoltà vengono meno.
Cominciamo analizzando un header fornito dalla COIN, che è per così dire "una libreria" poiché contiene tutte le costanti e tutte le macro di chiamata fatte per poter esportare il codice ed usarlo in C.
L'header in questione è Coin_C_defines.h e attraverso l'analisi del codice vedremo come viene sfruttato appieno il preprocessing e come sia importante definire le chiamate per ogni ambiente che potremmo trovarci di fronte, perché come già visto le cose cambiano anche solo da compilatore a compilatore.

```
#if defined (CBC_EXTERN_C)
#define COIN_EXTERN_C
#define COIN_NO_CLP
#endif
```
(1)

```
/* We need to allow for Microsoft */
#ifndef COINLIBAPI

#if defined(CBCCINTERFACEDLL_EXPORTS) || defined(CLPMSDLL)
#if defined (COIN_EXTERN_C)
#   define COINLIBAPI __declspec(dllexport)
#else
#   define COINLIBAPI __declspec(dllexport)
#endif
#   define COINLINKAGE __stdcall
#   define COINLINKAGE_CB __cdecl
#else
```
(2)

```
#if defined (COIN_EXTERN_C)
#   define COINLIBAPI extern "C"
#else
#   define COINLIBAPI
#endif
#   define COINLINKAGE
#   define COINLINKAGE_CB
#endif
```
(3)

cominciamo con l'analizzare i primi 3 punti:

(1) Verifica se prima della chiamata di questo header è stata definita la costante CBC_EXTERN_C, in caso affermativo si occupa di definirne altre due: una riguarda

l'esternalizzazione in C (come vedremo successivamente) mentre l'altra si occupa di definire una costante per avvertire che non stiamo lavorando in CLP, bensì in CBC(questo file viene utilizzato anche per l'interfaccia C per il CLP, già disponibile nella distribuzione).

(2) Come prima cosa verifica se è stata definita COINLIBAPI, se **non** è stata definita allora guarda se è stata definita CBCCINTERFACEDLL_EXPORTS(per la dll di CBC) oppure CLPMSDLL(per la dll di CLP). A questo punto se è definita COIN_EXTERN_C, la macro impone di sostituire nel codice __declspec(dllexport) ovunque sia presente COINLIBAPI, altrimenti sostituisce COINLINKAGE con __stdcall e COINLINKAGE_CB con __cdecl. (Da notare un piccolo errore nelle due define __declspec, sono identiche, ma per quel che riguarda il nostro lavoro non hanno nulla a che vedere)

(3) Se invece non sono stato definite né CBCINTERFACEDLL_EXPORTS né CLPMSDLL(che non ci riguarda) allora, se è stato definito COIN_EXTERN_C si occuperà di sostituire COINLIBAPI con extern "C" (di cui abbiamo già visto la spiegazione in precedenza).

Volendo esprimere in maniera un po' più semplice quanto fa questo pezzo di codice, si potrebbe pensare a qualcosa di questo tipo: "guarda se ha intenzione di esternalizzare il codice, in caso affermativo, è una dll? Bene, in questo caso sostituisci COINLIBAPI con __declspec(dllexport) e COINLINKAGE con __stdcall. In caso contrario, se non è una dll sostituisci COINLIBAPI con extern "C", mentre se non è esternalizzabile ma è una dll sostituisci unicamente COINLINKAGE con __stdcall.
Analizziamo ora il pezzo di codice seguente

```
#if defined (CBC_EXTERN_C)
/* Real typedef for structure */
class Cbc_MessageHandler;
typedef struct {
  OsiClpSolverInterface * solver_;
  CbcModel              * model_;
  Cbc_MessageHandler    * handler_;
  char                  * information_;
} Cbc_Model;
#else
typedef void Cbc_Model;
#endif
```

Qui viene unicamente ridefinita la classe Cbc_MessageHandler in funzione del fatto che si abbia o meno intenzione di esternalizzare al C il codice.

Terminata quest'analisi abbiamo tutti gli elementi per poter scrivere un'interfaccia verso il C. Potremmo fare il tutto nel solo file .cpp in quanto le macro possono essere scritte ovunque, ma il lavoro risulterà sicuramente più ordinato e soprattutto più portabile e riutilizzabile se separiamo la parte di preprocessing nell'header dal codice vero e proprio.
Partiamo quindi dalla definizione del file header (il .h), il primo passo è includere l'header di cui si è trattato fino a poco fa (Coin_C_defines.h) e così facendo siamo già sicuri che il nostro codice dispone degli strumenti per essere esportato, eventualmente aggiungeremo qualche aggiustamento nel proseguio. Consiglio inoltre di includerlo tra apici, quindi deve trovarsi nella stessa cartella dell'header che stiamo costruendo e dei file sorgente.
A questo punto, basta riportare le firme dei metodi che si vuole redefinire secondo un certo schema che ora verrà descritto(poiché alcune parole chiavi nella firma verranno sostituite dal codice preprocessato) e racchiudere il tutto nel seguente schema

```
#ifdef __cplusplus
  extern "C"{
```

```
#endif
<insieme delle firme dei metodi>
#ifdef __cplusplus
   }
#endif
```

Al solito ormai, l'extern "C" sta ad indicare che il codice che segue la parentesi graffa deve rispettare le regole di compilazione del C (i nomi pertanto non verranno decorati).
Vediamo ora come devono essere costruite le firme dei metodi, prendendone una come esempio:

```
COINLIBAPI int COINLINKAGE
Cbc_readMps(Cbc_Model * model,const char *filename)
```

dove questo metodo legge un modello da da file mps e lo carica nel Cbc_Model passato.
Le parole chiave a cui rivolgere la nostra attenzione sono COINLIBAPI e COINLINKAGE.
Grazie al file che abbiamo precedentemente accluso, queste verranno tradotte in fase di preprocessing con i loro sostituti che si trovano nell'header Coin_C_defines.h a seconda di quali definizioni abbiamo fatto(perché ricordiamo, ogni define era dentro un blocco ifdef, per cui se la relativa parola chiave non è stata definita il blocco NON viene preprocessato).
Se abbiamo fatto tutto a dovere, il metodo sarà esportato e dato che è incluso in un blocco extern "C" in fase di compilazione non subirà la decorazione dei nomi e sarà a tutti gli effetti chiamabile da codice C.
Analizziamo ora il file .cpp. Non c'è molto altro da fare, poiché il grosso del lavoro viene fatto in fase di preprocessing, ovviamente qui dovremo includere il nuovo header, preceduto da una definizione:

#define CBC_EXTERN_C

Esempio tratto dal file .cpp

```
COINLIBAPI int COINLINKAGE
Cbc_readMps(Cbc_Model * model,const char *filename)
{
  const char prefix[] = "Cbc_C_Interface::Cbc_readMps(): ";
//  const int  VERBOSE = 2;
  if (VERBOSE>0) printf("%s begin\n",prefix);
  if (VERBOSE>1) printf("%s filename = '%s'\n",prefix,filename);

  int result = 1;
  result = model->model_->solver()->readMps(filename);
  assert(result==0);

  if (VERBOSE>0) printf("%s return %i\n",prefix, result);
  return result;
}
```

In cui si può notare la firma (come già definita nell'header), l'uso di un array di char per la stringa, tipico del C (dato che in C++ esiste il tipo string) e l'uso di metodi in C++ come ad esempio readMps(filename) che è stato definito in OsiDylpSolverInterface.cpp.
Passiamo ora al codice C che deve richiamare il nostro elaborato, l'interfaccia ormai è terminata per cui ora basta utilizzarla.
Anche qui gli approcci sono molteplici, noi ne vedremo un paio poiché il codice contenuto nell'header Coin_C_defines.h ha un'elasticità tale da consentircelo.
Le possibilità che analizzeremo sono le seguenti:

(1) Abbiamo la dll contenente l'intera libreria CBC(vedremo poi come includere la nostra interfaccia)
   a. Usare la dll staticamente e legarla al nostro programma in fase di compilazione
   b. Usare la dll dinamicamente attraverso il costrutto LoadLibrary

(2) Abbiamo il nostro codice diviso nelle varie librerie, dovremo quindi preparare un makefile adeguato e ricostruire la distribuzione

(1)a. Usare la dll staticamente

E' il caso più semplice, dato che il nostro programma in C non ha alcun accorgimento particolare. Eccezion fatta per due righe di codice di cui ci occuperemo subito, per il resto possiamo richiamare normalmente i metodi come se fossero definiti in quel pezzo di codice. Sarà poi nostra cura in fase di compilazione legare la dll al programma.

```
/*************************************************/
/*                                               */
        #define CBCCINTERFACEDLL_EXPORTS
        #define COIN_EXTERN_C
/*                                               */
/*************************************************/
#include "Cbc_C_Interface.h"
```

Come già analizzato in precedenza, la prima define serve a far si che la macro sostituisca i nomi con le rispettive chiamate nel caso dobbiamo lavorare con dll(come è ovvio, visto che la stiamo definendo). La seconda viene definita già in un altro punto, ma non c'è motivo di preoccuparsi, poichè la definizione è identica e viene semplicemente sovrascritta (è stata aggiunta qui unicamente per chiarezza).
A questo punto possiamo includere l'header da noi definito, che al suo interno avrà una chiamata annidata a Coin_C_defines.h e verrà quindi innescato il preprocessing (quando andremo a compilare) e verranno sostituite tutte le costanti con le opportune chiamate.

(1)b. Usare la dll dinamicamente

Questo caso, non è stato da me analizzato usando una dll della distribuzione CBC bensì della COIN-MP(che è veramente molto simile, si può dire che è una sua estensione) e pertanto negli spezzoni di codice faremo riferimento a quest'ultimo. Possiamo fare tutto ciò perché come vedremo il metodo di utilizzo, sarà pari pari per il CBC.
Cominciamo con il dare un breve sguardo a coinmp.h, in particolare a

```
#if defined(_MSC_VER)
#ifdef SOLVER_EXPORT
#define SOLVAPI extern "C"  __declspec(dllexport)
#define SOLVFUNC
#else
#define SOLVAPI __declspec(dllimport)
#define SOLVFUNC
#endif
#else
#define SOLVAPI
#define SOLVFUNC
#endif
```

dove si vede quanto già visto prima, la define riguardante __declspec(dllexport). Questo era tutto quello che ci occorreva sapere al riguardo, infatti ora basta includere il file CoinMP.h (da cui è tratto lo spezzone) e in fase di compilazione avremo che questo pezzo di codice viene preprocessato.

Guardiamo ora il codice C per capire l'uso del costrutto LoadLibrary

```
#include "CoinMP.h"
#include <windows.h>

//<--------------------------------->
//DEFINIZIONE PROTOTIPI

typedef char* (CALLBACK* COINGETVERSIONSTR)(void);
typedef char* (CALLBACK* COINGETSOLVERNAME)(void);
typedef HPROB (CALLBACK* COINCREATEPROBLEM)(char*);
typedef int (CALLBACK* COINREADFILE)(HPROB, int, char*);
typedef char* (CALLBACK* COINGETSOLUTIONTEXT)(HPROB, int);
typedef double (CALLBACK* COINGETMIPBESTBOUND)(HPROB);

void getline(char line[], int nmax);

//DEFINIZIONE COSTANTI
#define BUFFERLENGTH 128
//<--------------------------------->

int main (int argc, const char *argv[])
{
    //<--------------------------------->
    //VARIABILI LOCALI
    char input[BUFFERLENGTH];
    HINSTANCE hDLL;
    COINGETVERSIONSTR version;
    COINGETSOLVERNAME solver;
    COINCREATEPROBLEM createproblem;
    COINREADFILE file;
    COINGETSOLUTIONTEXT solutiontext;
    COINGETMIPBESTBOUND mipbb;
    //<--------------------------------->
    //CARICAMENTO DLL
    printf("Caricamento libreria CoinMP.dll...\n");
    hDLL = LoadLibrary("CoinMP.dll");
    printf("...fatto\n");
    //<--------------------------------->
```

Analizziamo prima questo spezzone, in cui si nota fra gli include l'header di cui parlavamo poc'anzi e windows.h, che occorre per poter usare i costrutti dinamici sulla dll.
Prestiamo un secondo di attenzione ai tipi definiti e notiamo che hanno una forma standard, prendiamo ad esempio il primo:
typedef char* (CALLBACK* COINGETVERSIONSTR)(void);
e andiamo a vedere come era fatta la firma del metodo CoinGetVersionStr in CoinMP.h:
SOLVAPI char* CoinGetVersion(void);

A questo punto non ci vuole molto ad intuire che le due cose sono strettamente connesse, infatti, eccezion fatta per SOLVAPI che scomparirà in fase di preprocessing:

- Hanno lo stesso tipo di dato ritornato
- Hanno gli stessi parametri
- Il nome del metodo viene visto come puntatore di tipo CALLBACK

Vediamo come far uso di quanto appena ricavato. Creiamo un dato di tipo HISTANCE e attraverso il metodo LoadLibrary carichiamo in esso la dll. Creiamo inoltre altre istanze, una per ogni tipo di dato che abbiamo definito in precedenza.

```
//<--------------------------------->
//PER CHIAMARE UN METODO method=(cast) getProcAddress(hDll, "nome metodo");
//PER ESEGUIRE IL METODO RICHIAMATO method(lista parametri del metodo);

//DA QUI IN POI INIZIA IL WRAPPER

if(hDLL){
    //<--------------------------------->
    //METHOD CONTIENE IL METODO CHE VIENE RICHIAMATO ATTRAVERSO GETPROCADDRESS
    //IN QUESTO CASO LA VERSIONE DEL SOFTWARE COIN IN USO
    version = (COINGETVERSIONSTR) GetProcAddress(hDLL, "CoinGetVersionStr");
    printf("* Ambiente CoinMP * \n");
    printf("CoinMP.dll versione: %s \n", version());
    solver = (COINGETSOLVERNAME) GetProcAddress(hDLL, "CoinGetSolverName");
    printf("Risolutore: %s \n", solver());
```

A questo punto, basta solo capire come prelevare i metodi dalla libreria.
Cominciamo con un controllo sull'oggetto libreria, per vedere se è stata caricata correttamente; in caso affermativo attraverso il metodo GetProcAddress, che ha come parametri libreria e nome metodo, otteniamo l'indirizzo di memoria a cui si trova il metodo e con un semplice cast lo assegnamo al dato associato.
Ora basta utilizzare il dato stesso come un metodo, in questo caso senza parametri e abbiamo terminato.
Al termine dell'applicativo, la riga FreeLibrary(hDll); ci consente di rilasciare la risorsa.

(2)Usare il codice separatamente

Anche in questo caso, come l'(1)a., non abbiamo di che preoccuparci. Infatti basta includere il nostro header nell'applicativo C e possiamo lavorare tranquillamente, avendo l'unica accortezza di scrivere #define COIN_EXTERN_C prima di includere il file in maniera tale che questo preprocessi tutto il contenuto del .cpp in fase di compilazione.
Ci si occuperà delle altre problematiche in fase di creazione del makefile(necessario in questo caso).

L’applicativo

Presentazione

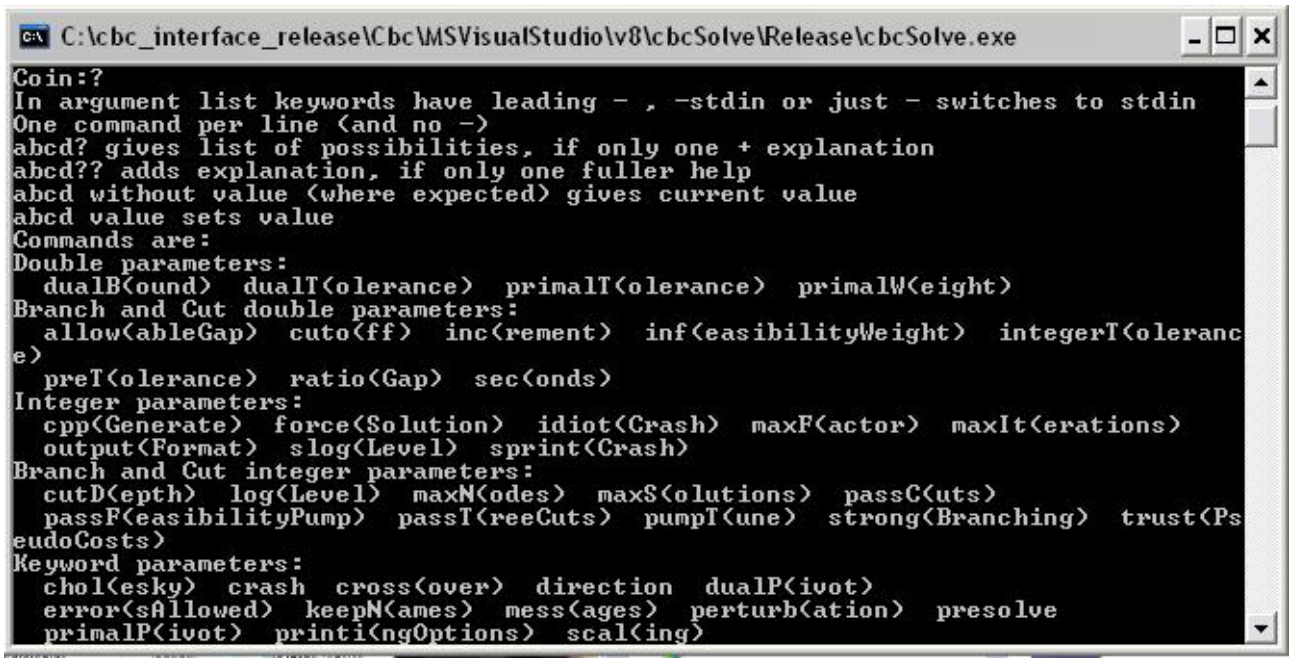
Presa finalmente confidenza con la distribuzione Cbc, il suo utilizzo ed un eventuale interfacciamento, passiamo ad una breve presentazione del mio elaborato (che può essere utilizzato a titolo di esempio).
Il primo passo, successivo alla creazione dell’interfaccia, è stato capire che cosa rendere disponibile in C per poter essere utilizzato, e cosa realmente utilizzare nel nostro applicativo. Infatti, è lecito pensare di rendere disponibili più metodi di quelli visibili all’utente finale, in maniera tale da favorire un ulteriore sviluppo.
In secondo luogo, ma non per questo meno importante, dovremo stabilire il modo in cui l’applicativo si relazionerà con l’utente; infatti le scelte sono molteplici e ovviamente ognuna avrà un modo diverso di essere codificata:

- Da linea di comando?
- Da un menù (anche se semplificato)
- Entrambe le cose?

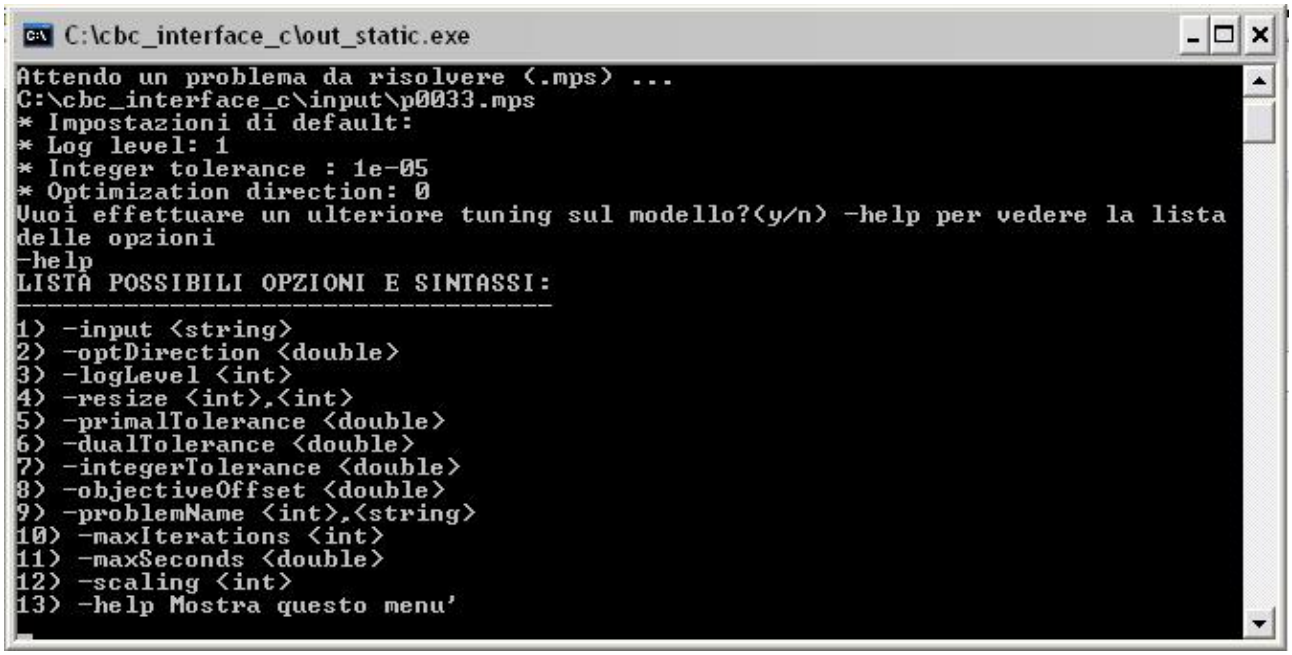
Soluzione

La scelta adottata è stata quella di ricalcare nella maniera più fedele possibile l’impostazione della COIN, come nell’immagine seguente:

facendo cioè in modo che se non vengono passati parametri da linea di comando, allora viene fornita un’interfaccia grafica con cui interagire.
Inoltre l’idea è quella di fornire un help(seppur in forma ridotta) con la lista e la sintassi dei comandi, in maniera analoga a quella di figura:



Da questa scelta, è nato un applicativo in grado di fare cose in maniera analoga seppur minore. Infatti, è possibile risolvere il problema e impostare alcuni parametri quali la direzione di ottimizzazione, lo scaling factor, la tolleranza intera; inoltre è stato aggiunto un help come in figura:



In conclusione è possibile:

- Eseguire il programma da riga di comando, con passaggio dei parametri (obbligatoriamente uno di questi dovrà essere il flag dell'input) – non viene effettuato alcun tipo di controllo sulla correttezza dei dati, quindi se sono sbagliati il programma crasha
- Eseguire il programma; in questo caso verrà presentato un menù da cui sarà possibile caricare un problema (contenuto in un file in formato mps) e risolvere il problema. A questo riguardo, sarà possibile:
    - Risolvere il problema mantenendo le impostazioni che sono ritenute di default

- Risolvere il problema effettuando un ulteriore tuning sul problema, al fine di migliorare le prestazioni

Metodi

Vediamo ora nel dettaglio quali metodi sono stati resi disponibili, accompagnati da una breve descrizione.

Tipo ritornato	Nome metodo	Descrizione
double	Cbc_getVersion()	Consente di ottenere la versione di CBC in uso
Cbc_Model*	Cbc_newModel()	Costruisce un'istanza di default di CbcModel
int	Cbc_readMps(Cbc_Model* model, const char* filename)	Legge un file mps, e scrive il suo contenuto nel modello che gli viene passato come parametro. Se il metodo ritorna 1, vuol dire che vi è stato un errore (file non esistente o errato)
void	Cbc_setPrimalTolerance(Cbc_Model* model, double value)	Imposta la tolleranza sul problema primale
void	Cbc_setDualTolerance(Cbc_Model* model, double value)	Imposta la tolleranza sul problema duale
void	Cbc_setIntegerTolerance(Cbc_Model* model, double value)	Imposta la tolleranza sul valore degli interi(cioè con che precisione si può accettare un risultato come intero)
void	Cbc_setObjectiveOffset(Cbc_Model* model, double value)	Imposta l'offset sul valore della funzione obiettivo
int	Cbc_setProblemName(Cbc_Model* model, int maxNchar, char* array)	Imposta il modello con un nome passato per parametro
void	Cbc_setOptimizationDirection(Cbc_Model* model, double value)	Stabilisce in che direzione fa ottimizzato il problema sapendo che: 1 – minimize -1 – maximize 0 – prende il default che viene dichiarato nell'mps
void	Cbc_setMaximumIterations(Cbc_Model* model, int value)	Imposta il numero massimo di iterazioni entro il quale deve terminare l'algoritmo, in

		caso non riesca termina ugualmente ma senza fornire risultato
void	Cbc_setMaximumSeconds(Cbc_Model* model, double value)	Analogo a quanto scritto sopra, solo in termine di secondi in questo caso
void	Cbc_scaling(Cbc_Model* model, int mode)	Imposta il fattore scaling, che può assumere uno dei seguenti valore: 0 – off 1 – equilibrium 2 – geometric 3 – auto (dipende da quanto scritto nell'mps) 4 – dynamic
void	printSolution(Cbc_Model* model)	Stampa a video
Cbc_Model*	getDefaultModel(char argv[])	Crea un modello dal file passato per parametro e risolve il rilassamento iniziale utilizzando le impostazioni di default
Cbc_Model*	getCustomModel(int argc, const char* parameters [])	Come sopra, solo questa volta prima di risolvere il rilassamento iniziale al modello vengono applicate le impostazioni che vengono passate mediante parametro
int	callOption(const char* parameters[], int index, Cbc_model* model)	Viene applicata l'opzione che si trova all'indice index e index +1 (eventualmente +2) al modello e si aggiorna l'indice sulla posizione dei parametri da analizzare

Compilazione

Introduzione

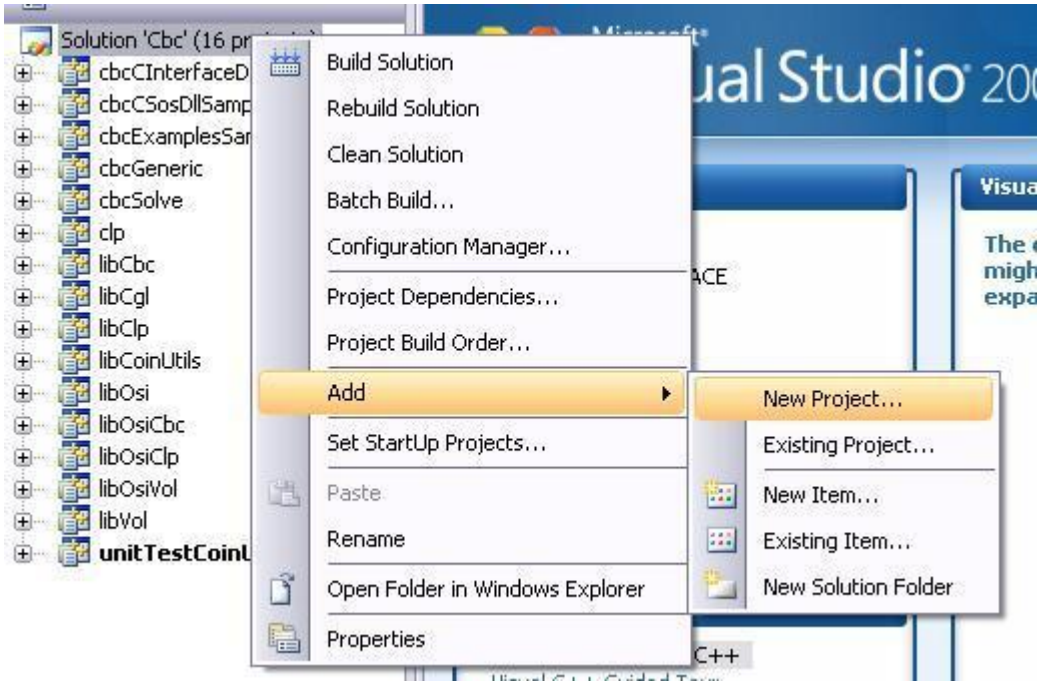
Siamo arrivati all'aspetto forse più complicato di tutto il percorso, almeno per me. Infatti qui le possibilità sono le più varie e nessuna particolarmente semplice, ma ora un passo alla volta vedremo di fare chiarezza.
Supporremo di trovarci a che fare con le seguenti situazioni:

- Ricompilazione della dll
- Compilazione con linking ad una libreria statica
- Compilazione con linking ad una libreria dinamica
- Guida ai flag di gcc
- Generazione di un makefile e ricompilazione

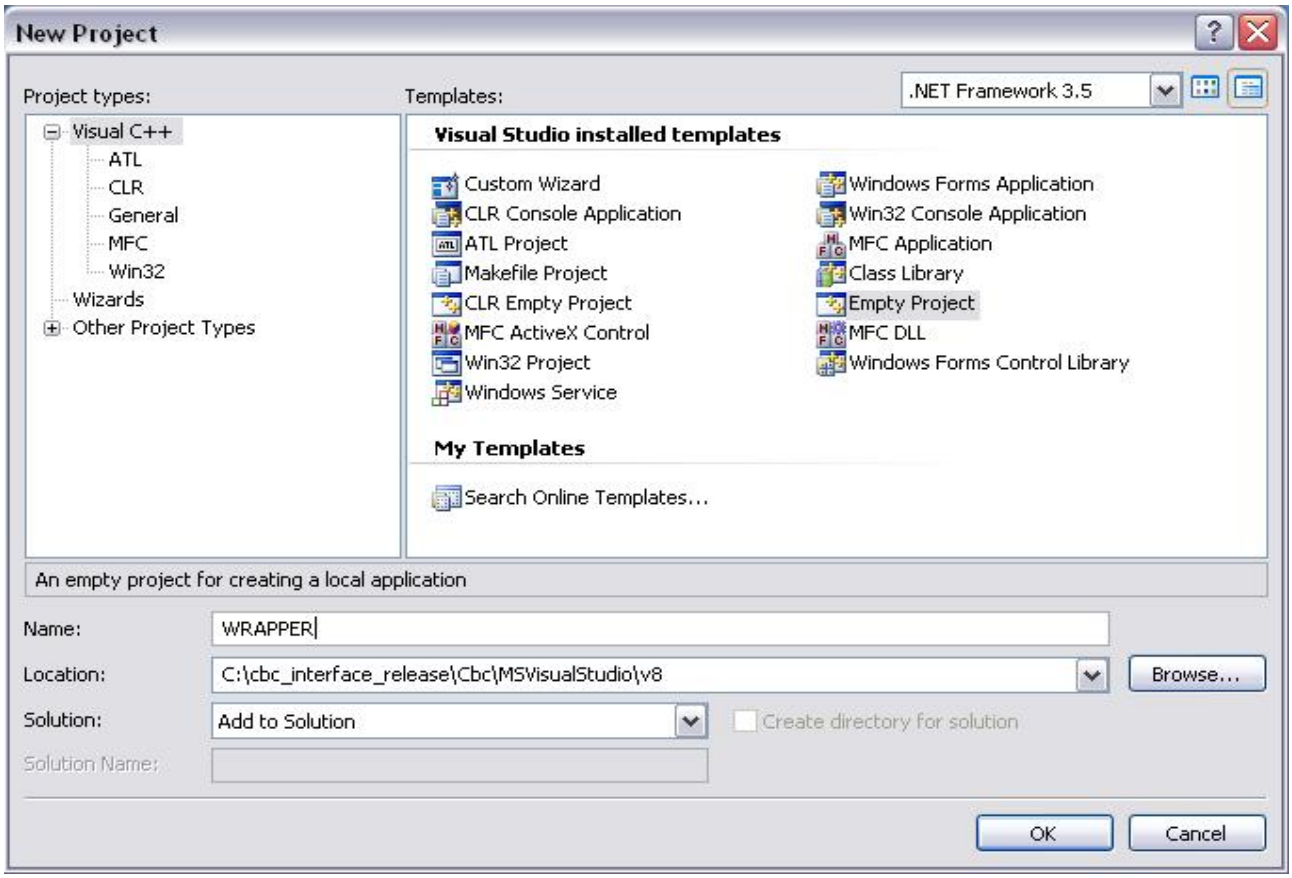
che sono quelle a cui abbiamo fatto riferimento fino ad adesso.

Ricompilazione della dll

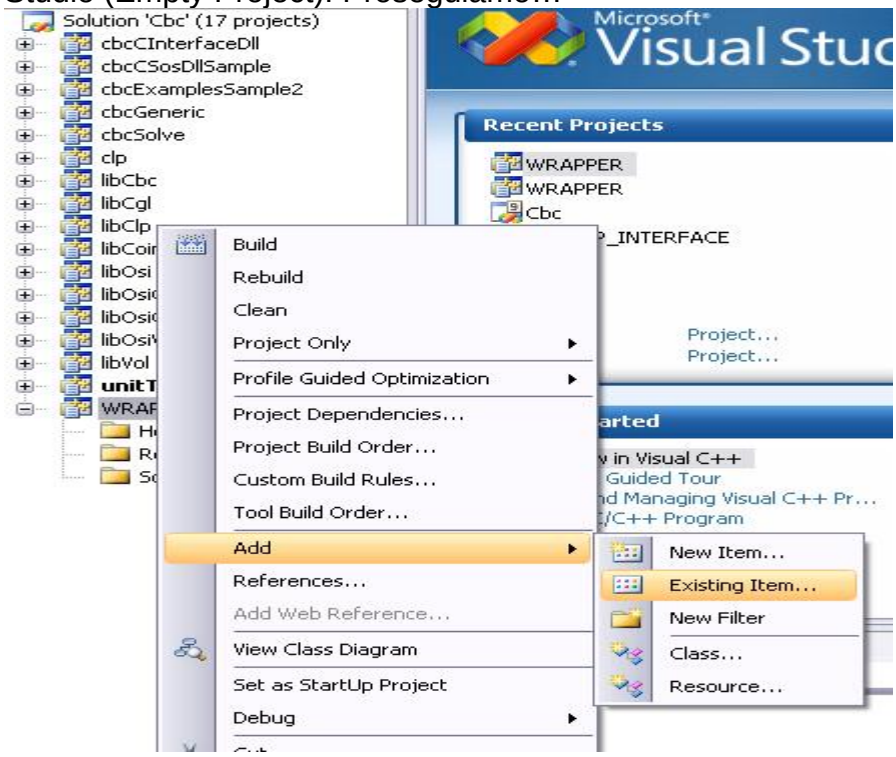
Quanto segue farà riferimento alla versione 9 del Visual Studio, ma in linea di massima il procedimento è sempre il medesimo; sarebbe possibile creare la dll anche in gcc, ma è una strada molto meno percorribile.
Infatti utilizzare VC++ in questo caso è molto più intuitivo, in aggiunta non dobbiamo preoccuparci di rispettare convenzioni né altro dato che se ne occupa lui.
Inoltre le impostazioni richieste all'utente come vedremo sono davvero poche facilitando ulteriormente il lavoro, e in output, oltre alla libreria dll otteremo anche .lib e .exp che come avremo modo di constatare sono utili in fase di compilazione.
Per prima cosa dobbiamo lavorare sulla soluzione che abbiamo utilizzato per compilare e creare la dll e a questa aggiungere un nuovo progetto come segue:



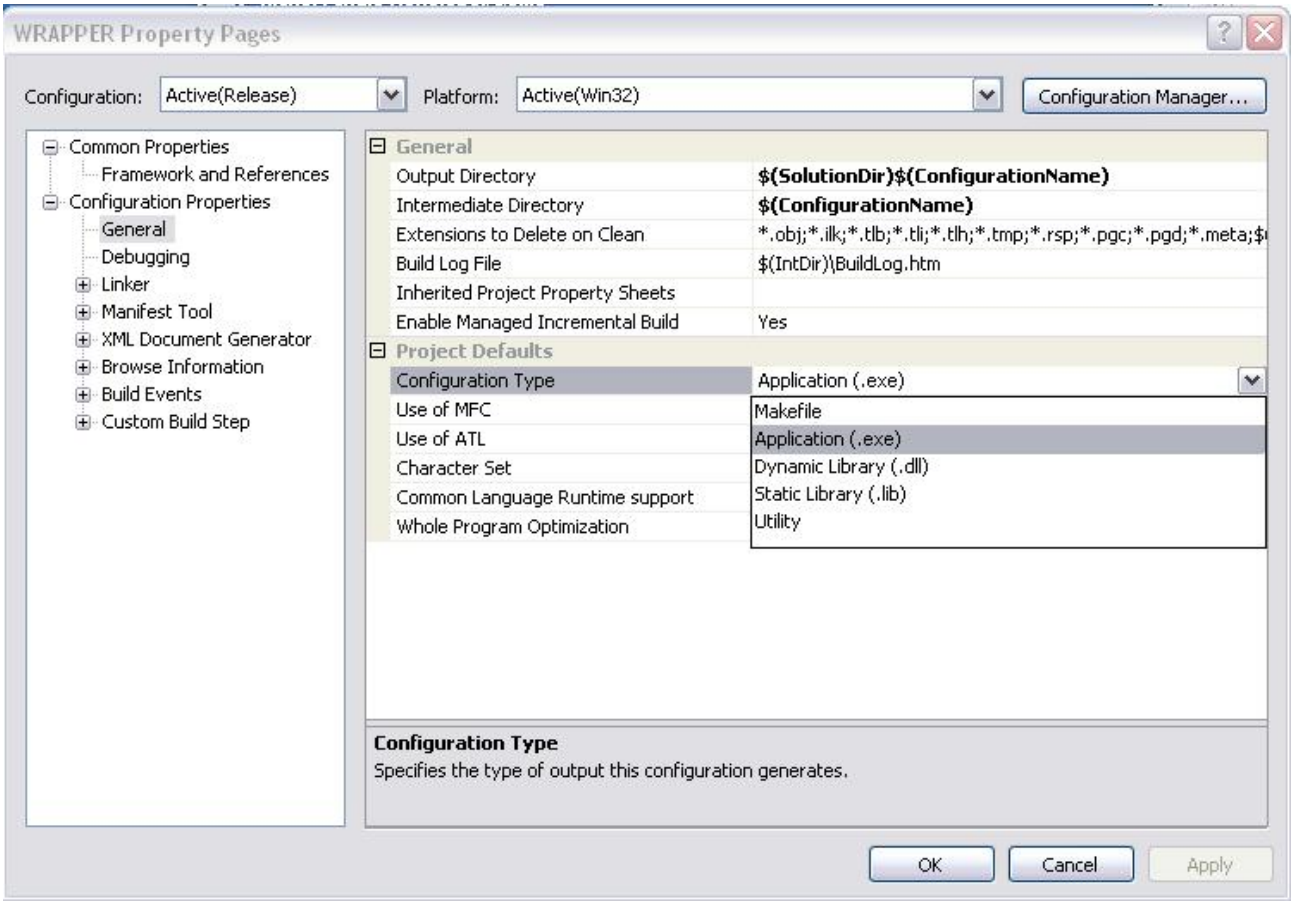
e con un semplice click ci troveremo qui:



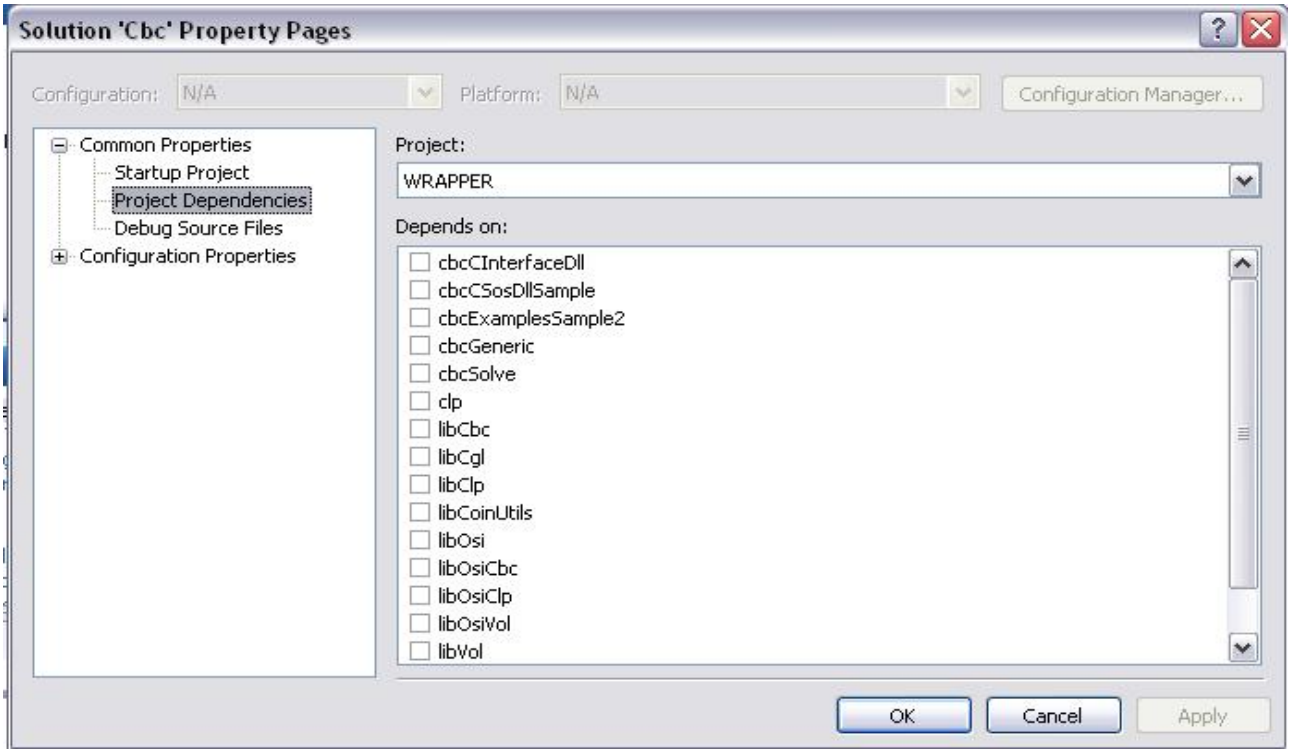
Notare le varie impostazioni: Add to Solution, la location, la scelta del template di Visual Studio (Empty Project). Proseguiamo…



Ora dobbiamo “addare” l’interfaccia (intesa come .cpp e .h, si occupa lui di separare codice e header) e Coin_C_defines.h, ricordando che il tutto deve risiedere nella medesima cartella(infatti è stato incluso con gli apici).
Ora, andiamo nella pagina delle proprietà del progetto (**non** la soluzione) e da qui impostiamo il tipo di output che desideriamo, nel nostro caso dll.



Inoltre è opportuno andare a controllare che abbia aggiornato la dipendenze fra i vari progetti e che abbia incluso la cartella contenente la nostra interfaccia nel percorso di compilazione; se per qualsiasi motivo non l'avesse fatto occorre farlo a mano seguendo l'esempio riportato in figura(ora su proprietà della soluzione):



A questo punto basta unicamente ricompilare (build) la distribuzione e VC++ si occuperà solamente di ciò che è stato aggiunto.
Tutto questo è stato fatto per avere una dll nuova che sostituisse la precedente, infatti questa è stata costruita sulla nostra interfaccia, permettondoci di avere una libreria chiamabile dal C che contenga tutto ciò che occorre e sia inoltre una “scatola chiusa”.

Ora possiamo occuparci di compilare il nostro programma C e di linkare la libreria, vediamo le possibilità:

Compilazione con linking ad una libreria statica

Per prima cosa apriamo la nostra shell Cygwin e ci posizioniamo nella cartella contenente i file sorgente, dove a questo punto si troveranno:

- Sorgenti in C
- Sorgenti in C++
- Header dell’interfaccia
- Coin_C_defines.h
- Libreria dll (prodotta da VC++)
- Libreria lib (prodotta da VC++)
- Libreria exp (prodotta da VC++)

A questo punto dobbiamo unicamente utilizzare il nostro gcc per compilare, nella seguente maniera:

```
gcc -static interfaccia_static.c -L. -lcbcCInterfaceDll -o out_static.exe
```

Vedremo nel seguito la spiegazione delle opzioni.

Compilazione con linking ad una libreria dinamica

Per prima cosa apriamo la nostra shell Cygwin e ci posizioniamo nella cartella contenente i file sorgente, dove a questo punto si troveranno:

- Sorgenti in C
- Sorgenti in C++
- Header dell’interfaccia
- Coin_C_defines.h
- Libreria dll (prodotta da VC++)
- Libreria lib (prodotta da VC++)
- Libreria exp (prodotta da VC++)

A questo punto dobbiamo unicamente utilizzare il nostro gcc per compilare, nella seguente maniera:

gcc interfaccia_dynamic.c –o out_dynamic.exe

Qui non occorre includere nessuna opzione particolare, dato che anche la chiamata alla libreria viene effettuata internamente al codice.

Guida ai flag di gcc

Dato che abbiamo utilizzato gcc per compilare, è opportuno fare un'analisi un po' più accurata dei flag utilizzati (nel caso di linkaggio verso libreria statica).

- -static aggiunge le opzioni necessarie per linkarsi con libdld.sl, forzando la compilazione statica del tutto
- -L. cerca nelle directory in ordine, partendo dalla radice
- -l include la libreria specificata di seguito, non deve avere estensione e devono essere presenti il file dll e il lib (su alcune guide è riportato che per poter funzionare richiede che le prime tre lettere della dll siano lib, ma sulla reference non si fa alcun riferimento a ciò)
- -o fornisce l'output sul file specificato

Per qualsiasi altra opzione, basta consultare la reference ma in linea di massima questo è quanto.

Generazione di un makefile e ricompilazione

Siamo all'ultimo caso, in cui si è scelto di mantenere separate le librerie e quindi sorge la necessità di generare un makefile adeguato da utilizzare mediante gcc (in ambiente Cygwin).
Prima di proseguire, vediamo brevemente cos'è quello di cui stiamo parlando:
make è una utility usata con i sistemi operativi della famiglia UNIX che automatizza il processo di conversione dei file da una forma ad un'altra, risolvendo le dipendenze e invocando programmi esterni per il lavoro necessario.
Molto frequentemente è usato per la compilazione di codice sorgente in codice oggetto, unendo e poi linkando il codice oggetto in esegubili o in librerie. Esso usa file chiamati "makefiles" per determinare il grafico delle dipendenze per un particolare output, e gli script necessari per la compilazione da passare alla shell.
Come riportato sui paper messi a disposizione dalla COIN **If you want to link your own code with a COIN library, it is probably best to start with this example Makefile;**
che riporto qui per poterlo commentare:

```
##############################################################################
#    You can modify this example makefile to fit for your own program.   #
#    Usually, you only need to change the five CHANGEME entries below.   #
##############################################################################

# CHANGEME: This should be the name of your executable
EXE = my_prog

# CHANGEME: Here is the name of all object files corresponding to the source
#           code that you wrote in order to define the problem statement
OBJS =  my_prog.o \
        additional_source1.o \
        additional_source2.o

# CHANGEME: Additional libraries
ADDLIBS = -L$(HOME)/MyLibrariesDir -ladditinal_lib

# CHANGEME: Additional flags for compilation (e.g., include flags)
ADDINCFLAGS = -I$(HOME)/MyCoolPrograms/include

# CHANGEME: Directory to the sources for the (example) problem definition
# files
SRCDIR = .
VPATH = .
```

Questa è la parte di nostro interesse, cioè quella che normalmente va modificata per poter compilare il nostro programma.

Le modifiche sono prevalentemente le seguenti:

- EXE è il nome dell'eseguibile che si vuole produrre con questo makefile.
- OBJS è la lista di tutti i file oggetto che corrispondo ai file sorgente che si vuole compilare.
- ADDLIBS è la lista di tutte le librerie **non** COIN richieste nel programma finale (e relativi flag).
- ADDINCFLAGS è l'insieme dei flag utilizzati per la compilazione che non sono standard per COIN (cioè non sono già inclusi).
- SRCDIR and VPATH vanno impostate con il percorso della directory in cui si trova il codice sorgente.

Quindi un possibile makefile potrebbe essere ad esempio:

- EXE = CBC_C.exe
- OBJS = <interfaccia_c>.o \
  <programma_c>.o
- ADDLIBS vuoto
- ADDINCFLAGS vuoto
- SRCDIR= directory di lavoro (contenente i file sorgente)
  VPATH = directory di lavoro (contenente i file sorgente)

Passiamo quindi al resto del makefile, ma prevelentemente per un'occhiata, dato che nella maggior parte dei casi questa parte rimarrà immutata.

```
##########################################################################
#  Usually, you don't have to change anything below.  Note that if you   #
#  change certain compiler options, you might have to recompile the      #
#  COIN package.                                                         #
##########################################################################

# C++ Compiler command
CXX = g++

# C++ Compiler options
CXXFLAGS = -O3 -fomit-frame-pointer -pipe -DNDEBUG  -pedantic-errors \
      -Wimplicit -Wparentheses -Wreturn-type -Wcast-qual -Wall \
      -Wpointer-arith -Wwrite-strings -Wconversion

# additional C++ Compiler options for linking
# Don't add rpath if you want to have rpms built from your package,
# since many distributions forbid it.
CXXLINKFLAGS =  -Wl,--rpath -Wl,/home/me/Coin-Pkg/lib

# Directory with COIN header files
COININCDIR = /home/me/Coin-Pkg/include

# Directory with COIN libraries
COINLIBDIR = /home/me/Coin-Pkg/lib

# Libraries necessary to link with Clp
LIBS = -L$(COINLIBDIR) -lCbc -lCgl -lOsiClp -lOsiCbc -lOsi -lClp -lCoinUtils \
        \
        `cat $(COINLIBDIR)/cgl_addlibs.txt` \
        `cat $(COINLIBDIR)/osi_addlibs.txt` \
        `cat $(COINLIBDIR)/clp_addlibs.txt` \
        `cat $(COINLIBDIR)/coinutils_addlibs.txt`
```

I nomi delle variabili dei compilatori CXX, CC e F77 sono già stati adattati al sistema come anche i flag CXXFLAGS, CFLAGS, FFLAGS. Questi sono gli stessi utilizzati per compilare la libreria COIN e quindi non è una cattiva idea usare le stesse opzioni.
COININCDIR e COINLIBDIR sono stati adattati alle directory dove sono stati installati gli header e le librerie COIN.
La variabile LIBS è impostata con i flag di compilazione occorrenti per linkarsi alla libreria COIN, ad esempio nello spezzone di codice sopra è la libreria CBC.

```
# Necessary Include dirs (we use the CYGPATH_W variables to allow
# compilation with Windows compilers)
INCL =  -I`$(CYGPATH_W) $(COININCDIR)` $(ADDINCFLAGS)

# The following is necessary under cygwin, if native compilers are used
CYGPATH_W = echo

all: $(EXE)

.SUFFIXES: .cpp .c .o .obj

$(EXE): $(OBJS)
       bla=;\
       for file in $(OBJS); do bla="$$bla `$(CYGPATH_W) $$file`"; done; \
       $(CXX) $(CXXLINKFLAGS) $(CXXFLAGS) -o $@ $$bla $(ADDLIBS) $(LIBS)

clean:
       rm -rf $(EXE) $(OBJS)

.cpp.o:
       $(CXX) $(CXXFLAGS) $(INCL) -c -o $@ `test -f '$<' || echo '$(SRCDIR)/'`$<


.cpp.obj:
       $(CXX) $(CXXFLAGS) $(INCL) -c -o $@ `if test -f '$<'; then $(CYGPATH_W) '$<'; else $(CYGPATH_W) '$(SRCDIR)/$<'; fi`

.c.o:
       $(CC) $(CFLAGS) $(INCL) -c -o $@ `test -f '$<' || echo '$(SRCDIR)/'`$<


.c.obj:
       $(CC) $(CFLAGS) $(INCL) -c -o $@ `if test -f '$<'; then $(CYGPATH_W) '$<'; else $(CYGPATH_W) '$(SRCDIR)/$<'; fi`
```

Nel caso si faccia uso di un compilatore Windows puro, oppure si lavori in ambiente Cygwin il pezzo di codice qui sopra è richiesto poiché la sintassi è differente da Unix (ma dovrebbe comunque essere già presente nel makefile).

A questo punto, terminata correttamente la procedura di creazione del makefile basta un semplice make e lui ricompilerà la parte di distribuzione non ancora compilata(quella da noi aggiunta) e fornirà l'output.

Sull'esecuzione

Qui c'è veramente poco da dire, se non che dipenderà tutto da come avete configurato il compilatore nella fase appena terminata.
Infatti lì avete specificato dove l'eseguibile dovrà cercare gli header o eventualmente le dll. Nel caso abbiate proceduto con un makefile avrete solo eseguibile e .h(nelle rispettive directory), mentre, se avete lavorato con gcc e dll avrete l'eseguibile nella medesima directory della libreria e dell'header dell'interfaccia. In entrambi i modi però, dovrete fare in modo di avere il file cygwin1.dll nella medesima cartella dell'eseguibile poiché è necessario per il funzionamento del vostro applicativo.

Conclusioni

Questo è quanto, la guida fornisce una prima presentazione del software Cbc ed un suo probabile utilizzo interfacciandosi con un altro linguaggio. Le cose anche se apparentemente semplici, non lo sono quasi mai state, perchè la documentazione al riguardo non è molta e soprattutto non molto dettagliata. Questo non credo sia tanto da imputare alla COIN, ma al fatto che comunque ci si aspetta che questo software venga utilizzato da persone che hanno conoscenze piuttosto avanzate di programmazione C++. Spero quindi, che questa mia guida possa aiutare qualcuno che pur non avendo molta disinvoltura con questo linguaggio voglia cimentarsi nell'impresa.
All'atto della stesura di questo elaborato, le informazioni contenute potrebbero risultare già "datate" a causa del continuo evolversi di questo software (proprio per via della sua natura Open Source); in ogni caso sicuramente fornisce una traccia su come bisogna muoversi per ottenere il più velocemente possibile una discreta padronanza di CBC, specie sugli aspetti più intricati.
Riassumendo:

- In fase di installazione della distribuzione seguire quanto detto precedentemente, ricordando che se si ricade in un qualche errore che qui non è stato riportato si può consultare il sito di cygwin, quello della COIN e Google o qualsiasi altro motore di ricerca in generale alla ricerca di informazioni, magari qualcuno si è già trovato nel vostro caso.
- Per quel che riguarda la teoria necessaria, oltre a quella richiesta di Ricerca Operativa e più nello specifico di programmazione lineare intera, sono necessarie conoscenze di programmazione C e C++. Tutto ciò che riguarda l'interconnessione tra questi due linguaggi è stato spiegato in questa relazione in maniera esaustiva anche attraverso esempi; oltre a questo, se non dovesse bastare, si può fare affidamento alle reference, a manuali e a materiale reperibile in Internet (anche se l'argomento viene trattato di rado e sempre in maniera approssimativa).

- Per ciò che riguarda la fase di compilazione, le diverse strade sono state spiegate ampiamente e mediante screenshot pertanto non credo ci possano essere dubbi di sorta.

Inoltre ricordiamo che Trac (il sito di riferimento del progetto, ce n'è uno di diverso per ogni distribuzione, in questo caso https://projects.coin-or.org/Cbc/wiki) è un utile strumento: permette di trovare risposte fra i "ticket" di altri che hanno avuto dubbi magari simili ai nostri, segnalato bug, approntato modifiche. In aggiunta a questo c'è la possibilità di iscriversi alla mailing list del progetto, mantenendo così il passo negli aggiornamenti. Magari capiterà che siamo noi stessi a segnalare qualche errore di cui ci siamo accorti, contribuendo così al miglioramento del software. E' proprio questa collaborazione parte della potenza di COIN.

Bibliografia

- The C Programming Language by Ritchie Kernighan
- The C++ Programming Language Special 3rd Edition
- http://www.wmlscript.it/cpp/page8.php
- http://forum.masterdrive.it/c-c-21/c-collegamento-esplicito-dll-22602/
- http://forum.masterdrive.it/c-c-21/win32-creare-richiamare-dll-12016/
- http://www.coin-or.org/Cbc/cbcuserguide.html
- http://www.imada.sdu.dk/~svalle/courses/dm14-2005/mirror/c/ccourse.html
- http://forum.masterdrive.it/c-c-21/c-c-differenze-h-dll-21922/
- http://www.adp-gmbh.ch/cpp/gcc/create_lib.html
- http://www.net-free.it/articoli/viewart.php?idart=96
- http://aelinik.free.fr/c/ch23.htm
- http://gcc.gnu.org/ml/gcc-help/2002-12/msg00088.html
- http://en.wikipedia.org/wiki/Name_mangling
- https://projects.coin-or.org/BuildTools/wiki/user-compile
- http://forum.html.it/forum/showthread.php?s=&threadid=1187404
- Gcc reference