

Strumenti per la programmazione C++ in Linux

g++, make, doxygen, cvs

Francesco Versaci

webmonster --apud-- apf.it

MonteLUG

Montebelluna Linux User Group



Licenza d'utilizzo

Copyright © 2005, Francesco Versaci.

Questo documento viene rilasciato secondo i termini della licenza Creative Commons (<http://creativecommons.org>).

L'utente è libero di:

distribuire, comunicare al pubblico, rappresentare o esporre in pubblico la presente opera

alle seguenti condizioni:

Attribuzione Deve riconoscere la paternità dell'opera all'autore originario.

Non commerciale Non può utilizzare quest'opera per scopi commerciali.

No opere derivate Non può alterare, trasformare o sviluppare quest'opera.

In occasione di ogni atto di riutilizzazione o distribuzione, deve chiarire agli altri i termini della licenza di quest'opera.

Se ottiene il permesso dal titolare del diritto d'autore, è possibile rinunciare a ciascuna di queste condizioni. Le utilizzazioni libere e gli altri diritti non sono in nessun modo limitati da quanto sopra. Questo è un riassunto in lingua corrente dei concetti chiave della licenza completa (codice legale), reperibile sul sito Internet

<http://creativecommons.org/licenses/by-nc-nd/2.0/legalcode>



Cosa vedremo?

Strumenti per la compilazione

- g++** Il compilatore C++ della GNU
- make** Per automatizzare la compilazione (e altro)

Documentazione e gestione dei sorgenti

- doxygen** Per automatizzare la documentazione dei sorgenti
- cvs** Per gestire e condividere gli archivi di sorgenti



g++ – Panoramica

Passi per la creazione dell'eseguibile

preprocessing Vengono espanso le macro
(fra cui le direttive `#include`)

compilation Il sorgente viene convertito in
linguaggio *assembly*

assembling Dall'*assembly* al linguaggio
macchina

linking Si uniscono i vari pezzi per
formare l'eseguibile finale

runtime Vengono caricate in memoria
le eventuali librerie condivise



g++ – Opzioni globali

Estensioni riconosciute

Le estensioni riconosciute dal compilatore come file C++ sono le seguenti:

header .h

codice .cc .C .cpp .cxx .cp

Opzioni per l'output

```
g++ [-c] [-o output] nome.cc
```

-c Compila senza collegare, non crea l'eseguibile

-o *nomeoutput* Chiama il file di uscita "nomeoutput"



g++ – Esempio di compilazione base 1/2

saluti.h

```
#ifndef SALUTI_H
#define SALUTI_H

#include <iostream>
using namespace std;

void ciao ();

#endif
```

saluti.cc

```
#include "saluti.h"
void ciao(){
    cout << "ciao!\n";
}
```

test.cc

```
#include "saluti.h"
int main(){
    ciao ();
    return 0;
}
```

g++ – Esempio di compilazione base 2/2

```
g++ -c saluti.cc
g++ -c test.cc
g++ -o runme saluti.o test.o
./runme
ciao!
```

Spiegazione

- I primi due comandi compilano i file sorgenti creando i file oggetto `.o`.
- Il terzo li collega creando l'eseguibile `runme`, lanciato alla quarta riga.

Le prime tre righe si sarebbero potute sostituire con questa:

```
g++ -o runme saluti.cc test.cc
```



g++ – Opzione di avviso

Queste opzioni segnalano in fase di compilazione delle probabile sviste del programmatore che non danno errori in compilazione.

È consigliabile usarle **sempre**.

Warnings

- Wall** Abilita molti avvisi utili (variabili usate senza inizializzazione, classi polimorfe senza distruttore virtuale, ecc.)
- pedantic** Avvisa se si devia dallo standard ISO
- W (-Wextra)** Abilita altri avvisi (confronto di unsigned con 0, funzioni che possono o meno restituire un valore, ecc.)



g++ – Opzioni di debug

Le seguenti opzioni generano dati aggiuntivi per facilitare il lavoro a debugger e analizzatori vari.

Debug

- g Crea informazioni per debug per vari programmi
- ggdb Crea informazioni per debug con *gdb*
- pg Crea informazioni per *gprof*
- fprofile-arcs Registra le svolte prese nei salti condizionali, usato da *gcov*
- ftest-coverage Altre informazioni usate da *gcov*



g++ – Opzioni di ottimizzazione

Il g++ può utilizzare diverse tecniche per ottimizzare l'eseguibile prodotto.

Opzioni per l'ottimizzazione

- O1 Ottimizza salti condizionali e cicli, prova a eliminare alcuni if, srotola i cicli, ...
- O2 Ottimizza l'uso di sottoespressioni, riordina il codice minimizzando i salti, ...
- O3 Rende inline le funzioni piccole, ...



g++ – Scelta dell'architettura

Il compilatore permette inoltre di scegliere il codice assembly da produrre per specifici processori.

Quelli disponibili per i pc di casa sono i seguenti:

Architetture x86 disponibili

`-march=cpu-type` dove “cpu-type” è scelto frai seguenti:
i386, i486, i586, i686, pentium, pentium-mmx,
pentiumpro, pentium2, pentium3, pentium4,
prescott, nocona, k6, k6-2, k6-3, athlon,
athlon-tbird, athlon-4, athlon-xp, athlon-mp,
winchip-c6, winchip2 e c3



g++ – Opzioni di preprocessing

Le seguenti opzioni riguardano l'uso degli header.

Architetture x86 disponibili

- ldir* Aggiunge la directory “dir” al percorso di ricerca degli header
- MM* Produce in output le dipendenze dei file da inserire nel *Makefile*

Per esempio:

```
g++ -MM *.cc  
saluti.o: saluti.cc saluti.h  
test.o: test.cc saluti.h
```



g++ – Il collegamento

Opzioni di linking

- l*nome*** Usa la libreria esterna “nome”
- L*dir*** Cerca le librerie nelle directory “dir”
- shared** Crea un file oggetto condiviso, da usare come libreria dinamica
- static** Collega tutte le librerie in modo statico (anche le condivise)
- s** Togli la tavola dei simboli dall'eseguibile

Il comando ldconfig

Il linker guarda di default nelle directory di sistema (/lib, /usr/lib). Se si vogliono aggiungere altre directory senza specificarle ogni volta col parametro `-L` è necessario aggiungerle nel file `/etc/ld.so.conf` e lanciare il comando `ldconfig`

g++ – Compilazione con libreria esterna

test.cc

```
#include <gmpxx.h>
#include <iostream>
using namespace std;

int main(){
    mpq_class a("2/5");

    cout << a*a << endl;

    return 0;
}
```

Compilazione

```
g++ -o runme test.cc -lgmp -lgmpxx
./runme
4/25
```

Per usare una libreria esterna è necessario installare il pacchetto con gli header (*libqualcosa-dev*), includere l'header e dichiarare al linker in quali librerie recuperare gli oggetti (si veda il manuale della libreria usata).



make – Panoramica

Per automatizzare la compilazione dei sorgenti lo strumento standard è il `make`. Noi analizzeremo la versione GNU di questo programma.

Il Makefile

In ogni directory che contiene sorgenti è necessario creare un file di testo, chiamato *Makefile*, che contiene le istruzioni per la compilazione.

Come avviare la compilazione

Una volta creato il Makefile è sufficiente lanciare il comando `make` (o `make obiettivo`) per avviare la compilazione.



make – Regole

Sintassi delle regole

```
obiettivo: prerequisiti  
          comando1  
          comando2  
          ...
```

Spiegazione

- obiettivo** Il file da creare
- prerequisiti** I file che servono per crearlo
- comando** I comandi (si noti il *tab* obbligatorio) per creare l'*obiettivo*

Cosa fa il make?

Controlla se qualcuno frai *prerequisiti* è stato modificato piú recentemente dell'*obiettivo*. In caso affermativo esegue i *comandi*.



make – Un esempio banale

Makefile

```
grosso.txt: piccolo1.txt piccolo2.txt
    cat piccolo1.txt piccolo2.txt > grosso.txt
```

Il comando `make grosso.txt` controlla che il file *grosso.txt* non esista oppure sia piú vecchio dei file *piccolo1.txt* e *piccolo2.txt*. Quindi esegue il comando `cat` che crea il file *grosso.txt*.

Nel caso l'obiettivo sia già aggiornato `make` notifica la cosa con la seguente stringa:

```
make: 'grosso.txt' is up to date.
```



make – Phony targets

Spesso è comodo poter creare dei comandi svincolati da particolari file. Si usano allora degli obiettivi *phony*.

Makefile

```
.PHONY: clean  
clean:  
    rm *.o
```

Spiegazione

Eeguire il comando `make clean` è equivalente ad eseguire `rm *.o`



make – Variabili

Nei Makefile è possibile utilizzare delle variabili per evitare di riscrivere piú volte gli stessi comandi.

Makefile

```
CXXFLAGS = -Wall -W -pedantic
test.o : test.cc
    g++ $(CXXFLAGS) -c test.cc
saluti.o : test.cc
    g++ $(CXXFLAGS) -c saluti.cc
```



make – Regole implicite e variabili automatiche

Spesso si deve eseguire lo stesso comando su piú file.

Makefile

```
CXXFLAGS = -Wall -W -pedantic
%.o : %.cc
    g++ $(CXXFLAGS) -c -o $@ $<
```

Regole implicite

La seconda riga fornisce una regola per trasformare qualunque file `.cc` in un file `.o`

Variabili automatiche

- `$@` Bersaglio
- `$<` Primo prerequisito
- `^` Tutti i prerequisiti



make – Inclusione di altri file

È possibile includere altri file nel Makefile, cosa molto utile per includere file generati automaticamente.

Makefile

```
dep.mk: *.cc *.h
        g++ -MM *.cc > dep.mk

include dep.mk
```

Spiegazione

Il file *dep.mk* viene rigenerato ogni volta che si modifica un file sorgente. Viene incluso ad ogni make per fornire le dipendenze dei file.



make – Un esempio completo

```
CXXFLAGS = -Wall -W -pedantic
```

```
%.o : %.cc
```

```
g++ $(CXXFLAGS) -o $@ -c $<
```

```
runme : test.o saluti.o
```

```
g++ $(CXXFLAGS) -o $@ $^
```

```
dep.mk : *.cc *.h
```

```
g++ -MM *.cc > dep.mk
```

```
include dep.mk
```



make – Un esempio ricorsivo

```
# elenco directory in cui eseguire il make
DIRS =  dir1 dir2 dir3 \
        dir4 dir5 dir6

# esegui make in tutte le directory
.PHONY: all $(DIRS)

all:    $(DIRS)
        @echo 'Fatto!'

$(DIRS):
        $(MAKE) -C $@
```



doxygen – Panoramica



È uno strumento di documentazione per codice C/C++ e simili.

Come funziona?

- Si crea un file di configurazione
- Si commenta direttamente il codice (con una semplicissima sintassi)
- Si esegue il comando `doxygen`

Documentazione prodotta

La documentazione si può generare nei formati html, \LaTeX , pdf, ps e rtf.



doxygen – Configurazione

Doxyfile

Il file di configurazione, chiamato tipicamente *Doxyfile*, regola le diverse impostazioni (linguaggio, lingua, formati di documentazioni da produrre con le diverse opzioni, ...). Per creare un file di esempio, da modificare poi con un editor, basta digitare il comando `doxygen -g`.

doxywizard

Per i pigri esiste anche un frontend grafico che permette di configurare tutto da simpatici menu.

Creare la documentazione

Una volta creato il file di configurazione e commentati i sorgenti basta lanciare il comando `doxygen` perché si crei una directory *docs* contenente la documentazione.

doxygen – Commenti alle classi

mucca.h

```
/// Una mucca
/**
 * Simpatica mammifera erbivora
 * produttrice di latte
 */
class Mucca{
  private:
    int quantita_latte; ///< scorte disponibili
  public:
    /// mungi un po' di latte
    void mungi(int litri_latte);
    /// mangia un po' d'erba
    void mangia(int kg_erba);
};
```



doxygen – Documentare una funzione membro

C'è una sintassi particolare per commentare i parametri delle funzioni:

Esempio

```
/// Si informa sui gusti della mucca  
/**  
 * \param nome_erba Il nome dell'erba  
 * \return Si' o no?  
 */  
bool ti_piace(string nome_erba);
```



doxygen – Documentazione fuori dalle classi

head.h

```
/** \file head.h
 * \brief Qualche definizione
 */

/** \var typedef vector<int> vint;
 * \brief Vettore di interi
 */
typedef vector<int> vint;

/** \fn void saluta(string nome);
 * \brief Saluta i conoscenti
 * \param nome Il nome dell'amico
 */
void saluta(string nome);
```

Spiegazione

Per commentare cose fuori dalle classi è necessario innanzitutto aggiungere un commento al file header. Poi si possono aggiungere i commenti per le singole funzioni, variabili, ecc.



doxygen – Cosa produce il doxygen?

Interfacce

Viene creata una pagina per ogni classe, con le descrizioni dei relativi membri.

Diagrammi ereditarietà

Il doxygen genera anche dei diagrammi che illustrano le relazioni di ereditarietà.

Indici

Il documento prodotto contiene indici di classi, funzioni, header, esempi, ecc.

Link

Nel caso il formato d'output lo consenta vengono creati link alle definizioni ogni volta che sia possibile.



cvs – Panoramica

Il cvs è uno strumento per gestire archivi di sorgenti

Concorrenza

È studiato per affrontare i problemi di concorrenza che possono derivare da modifiche contemporanee di diversi sviluppatori

Accessibilità

Si può installare come server su una porta tcp ed essere accessibile in internet

Versioni e ramificazioni

Gestisce ramificazioni e fusioni di diverse versioni dei sorgenti. Mantiene in archivio tutte le modifiche fatte, per poterle rivedere ed eventualmente correggere.



cvs – Creazione del repository

init

Per creare un archivio è necessario scegliere una directory (per es. `/var/lib/cvs`) scrivibile dall'amministratore del cvs e dare il comando:

```
$cvs -d /var/lib/cvs init
```

che creerà la cartella

`/var/lib/cvs/CVSROOT` contenente i file di configurazione, che andranno modificati (quasi tutti) usando il cvs stesso.

Debian

In Debian l'inizializzazione dell'archivio è di solito fatta automaticamente (dietro richiesta) in fase di installazione.



cvs – Creazione di un progetto

import

Per inserire un nuovo progetto è sufficiente entrare nella directory contenente i sorgenti (o vuota se il progetto è nuovo) e lanciare il comando:

```
cvs -d /var/lib/cvs import dir_repository  
nome_autore tag_versione
```

Per es.

```
cvs -d /var/lib/cvs import gnu/gcc gnu start
```

creerà una directory `/var/lib/cvs/gnu/gcc` contenente il progetto gcc e associerà il tag `start` a tutti i file importati, in modo da poterli identificare con un nome.



cvs – Richiesta dei sorgenti

checkout

Per scaricare una copia locale di un progetto si usa il comando:
`cvs -d /var/lib/cvs co progetto`
che crea una directory `progetto` contenente l'ultima versione dei sorgenti.

Lo sviluppatore modificherà la copia locale e quando avrà apportato le modifiche le propagherà nel repository centrale.

CVSROOT

Impostando la variabile d'ambiente `CVSROOT` non c'è bisogno di specificare ogni volta il parametro `-d /var/lib/cvs`:
`export CVSROOT=/var/lib/cvs`



cvs – Aggiunta e rimozione di file

add

Per aggiungere un file al repository è necessario innanzitutto creare il file localmente e poi dare il seguente comando:

```
cvs add files...
```

Il file non viene aggiunto subito al repository remoto, ma viene solo schedulato.

rm

Per rimuovere un file si procede in modo simile: prima si cancella localmente, poi si schedula la rimozione:

```
cvs rm [files]
```

Si possono anche fare le due cose con un solo comando:

```
cvs rm -f files...
```



cvs – Modifica dei sorgenti

update

Dopo aver lavorato sulla copia locale si può vedere quali file si siano modificati e quali siano stati creati (andranno aggiunti esplicitamente all'archivio). Il comando da dare è:

```
cvs up [files]
```

Il comando `update` serve anche a rigenerare i file che sono stati cancellati.

Esempio

```
cvs add Makefile  
cvs rm -f test.cc  
vim access.cc  
cvs up  
cvs update: Updating .  
M access.cc  
A Makefile  
R test.cc
```



cvs – Propagazione delle modifiche

`commit`

Quando si è convinti delle modifiche che si sono fatte e che sono state rilevate dall'`update` si può concludere la sessione aggiornando l'archivio centrale nel seguente modo:

```
cvs ci [files]
```

In seguito al lancio del comando si apre un editor in cui scrivere una nota su ciò che si è modificato. Alternativamente si può specificare con il parametro `-m 'nota cambiamento'`.



cvs – Risoluzione dei conflitti

Gestione dei conflitti

Può succedere che qualcun altro abbia modificato il file su cui stiamo lavorando e propagato le modifiche prima di noi. In questo caso al momento dell'`update` le modifiche, se riguardano righe diverse del file, vengono fuse tranquillamente nella copia locale.

Se invece ci sono dei conflitti vengono scritte nella copia locale entrambe le versioni, lasciando allo sviluppatore la scelta su quale tenere (cancellando l'altra).

Cervisia

Possono essere molto comodi per la risoluzione dei conflitti dei tools grafici, quali ad esempio il *cervisia* o il *gcvcs*.



cvs – Esempio di conflitto

Versione 1

```
#include <iostream>
using namespace std;
int main(){
    cout << "Buongiorno\n";
    return 0;
}
```

Versione 2

```
#include <iostream>
using namespace std;
int main(){
    cout << "Buonasera\n";
    return 0;
}
```

Versione fusa

```
#include <iostream>
using namespace std;
int main(){
<<<<<<< main.cc
    cout << "Buongiorno\n";
=====
    cout << "Buonasera\n";
>>>>>>> 1.4
    return 0;
}
```



cvs – Gestione delle versioni

tag

Per assegnare un nome ad una versione si usa il comando *tag*:

```
cvs tag nome-release
```

Uso dei nomi simbolici

Il nome assegnato si può usare per distinguere le diverse versioni di un programma:

```
cvs co -r nome-release nome-progetto  
cvs up -r nome-release
```



cvs – Ramificazione

È spesso utile ramificare i sorgenti, per es. per continuare a mantenere una vecchia versione stabile di un programma mentre si continua a sviluppare quella instabile.

Creare una ramificazione

Il comando per creare una nuova ramificazione è:

```
cvs tag -b nome-branch
```



cvs – Fusione fra versioni

Se si mantiene una versione sperimentale dei sorgenti può essere utile fondere le modifiche nella versione stabile.

Fusione

Se la ramificazione sperimentale si chiama *test-branch-1*, per propagare le modifiche nella versione a cui si lavora è sufficiente dare il seguente comando:

```
cvs up -j test-branch-1
```

Lo sviluppatore viene avvisato delle fusioni fatte e degli eventuali conflitti.



cvs – Visione dei sorgenti

export

Per visualizzare i sorgenti senza volerli modificare si usa il comando *export*:

```
cvs export -r nome-release
```

oppure

```
cvs export -D data
```

dove *data* può essere anche “now”.



cvs – Confronto fra diverse versioni

diff

Per visualizzare le differenze fra diverse versioni di un file si usa il comando:

```
cvs diff -r nome-release nome-file
```

Anche in questo caso può essere molto comodo usare un tool grafico come *cervisia*, *gcvs* o *cvsmgdiff*.



cvs – Repository remoti con account di sistema

RSH

Per collegarsi ad un repository remoto ospitato in un server su cui si possiede un account si può usare l'rsh, nel seguente modo:

```
cvs -d :ext:utente@server:/var/lib/cvs co progetto
```

SSH

Per specificare una diversa shell remota si può impostare la variabile d'ambiente CVS_RSH:

```
export CVS_RSH=/usr/bin/ssh
```



cvs – Repository senza account di sistema – 1/2

PSERVER

Per non dare un account di sistema ad ogni sviluppatore il cvs fornisce un sistema di autenticazione autonomo: il *pserver*.

Configurazione

`/etc/services` Si aggiungono le seguenti righe:

```
cvspserver 2401/tcp
cvspserver 2401/udp
```

`/etc/inetd.conf` Si aggiunge la riga:

```
cvspserver stream tcp nowait
root /usr/sbin/tcpd cvs -f
--allow-root=/var/lib/cvs pserver
```

Debian La configurazione viene fatta automaticamente



cvs – Repository senza account di sistema – 2/2

CVSROOT/config

Per disabilitare l'autenticazione di sistema aggiungere `SystemAuth=no`. In Debian è necessario disabilitare anche l'autenticazione PAM: `PamAuth=no`

CVSROOT/passwd

Contiene gli utenti che hanno accesso al repository, l'hash delle loro password e l'utente di sistema che useranno per accedere all'archivio.

```
anonymous::cvsuser  
melissa:tGX1fS8sun6rY:cvsuser
```

CVSROOT/writers

Contiene la lista degli utenti che hanno permesso di scrittura sull'archivio

makepasswd

Per generare l'hash delle password si può usare il pacchetto *makepasswd*

cvs – Permessi di accesso ai progetti

Per poter accedere ad un progetto l'utente di sistema che rappresenta l'utente cvs deve avere permessi di scrittura nella directory del repository del progetto.
È quindi in generale buona cosa restringere l'accesso alla directory CVSROOT.



cvs – Sessione di esempio *pserver*

```
export CVSROOT=:pserver:utente@server:/var/lib/cvs
cvs login
cvs co progetto
...
cvs up
cvs ci
cvs logout
```



cvs – pserver su tunnel SSL – 1/3

Il brutto del *pserver* è che passa le **password in chiaro**.

Stunnel4 su Debian – Server – 1/2

- `apt-get install stunnel4`
- Eseguire tutte le procedure descritte in
`/usr/share/doc/stunnel4/README.Debian`
- Aggiungere al file `/etc/services` la riga:
`cvspservers 22401/tcp`
- `update-inetd --add 'cvspservers stream tcp
nowait root /usr/sbin/stunnel4 stunnel4
/etc/stunnel/stunnel-cvs'`



cvs – pserver su tunnel SSL – 2/3

Stunnel4 su Debian – Server – 2/2

- Rinominare `/etc/stunnel/stunnel.conf` in `stunnel-cvs`
- Impostare in `stunnel-cvs` le variabili `cert` e `key` col certificato creato prima
- Impostare `client=no`
- Togliere tutti i servizi e aggiungere le righe:
`exec=/usr/sbin/tcpd`
`execargs=/usr/sbin/cvs-pserver`



cvs – pserver su tunnel SSL – 3/3

Stunnel4 su Debian – Client

- Editare `/etc/default/stunnel4` impostando `ENABLE=1`
- Editare il file `/etc/stunnel/stunnel.conf` e aggiungere il servizio:

```
[cvspservers]
accept = 2401
connect = server-remoto:22401
```

- `/etc/init.d/stunnel4 restart`
- A questo punto si può usare il cvs come se fosse installato localmente:

```
export
CVSROOT=:pserver:utente@localhost:/var/lib/cvs
cvs login
...
```



cvs – Tools aggiuntivi





`cvsutils`

Permettono di effettuare alcune operazioni senza accedere alla repository remoto.

- `cvsu` Simula l'update in locale
- `cvschroot` Permette di cambiare al volo il repository
- `cvspurge` Elimina i file non nel repository (risultati della compilazione, ...)
- `cvsdo` Simula i comandi cvs in locale



Letture consigliate per approfondire...

-  **R. Stallmann, Comunità sviluppatori GCC**
Using the Gnu Compiler Collection
[info gcc](#)
-  **Autori vari**
Manuale make
[info make](#)
-  **Dimitri van Heesch**
Manuale doxygen
</usr/share/doc/doxygen/html/index.html>
-  **Roland Pesch e altri**
Manuale cvs
[info cvs](#)

