Sviluppo in C++

strumenti in ambiente Linux

(2014.08)

Indice

Introduzione

In questi appunti viene trattata una introduzione all'uso di alcuni strumenti, esistenti in ambiente Linux, utilizzabili per lo sviluppo di programmi in C++. Più specificatamente si tratterà di ***GNU Emacs*** editor nonché vero e proprio ambiente di sviluppo all'interno del quale possono essere utilizzati e integrati altri strumenti utili per la programmazione come quelli trattati in questi appunti, ***GCC*** (GNU Compiler Collection) suite di compilatori comprendente, fra l'altro, anche un compilatore C++, ***GDB*** (GNU Debugger) il *debugger*, ambiente per la correzione degli errori di programmazione. Si tratta, in tutti i casi, di software GPL sviluppati all'interno del progetto GNU dalla FSF di Richard Stallman (Free Software Foundation, http://www.fsf.org). La sigla GNU è un acronimo ricorsivo (Gnu's Not Unix) ed è associata ad un progetto che prevedeva, in origine, anche lo sviluppo di un intero sistema operativo. In seguito Linus Torvalds, il creatore di Linux, rilasciò il suo prodotto con licenza GPL (spesso si parla di GNU/Linux) e la fondazione ha di fatto abbandonato l'idea di un nuovo S.O. concentrandosi sullo sviluppo di strumenti ed utilità che fossero da corredo al S.O. stesso.

Il disegno riportato come intestazione di questi appunti è il logo del progetto.

Il processo di sviluppo

Scelto un linguaggio di programmazione (per esempio C o C++) e codificato un programma, il problema che si pone è quello di fare diventare tale programma eseguibile da un computer. A tale scopo è necessario utilizzare un ***compilatore*** (nel caso specifico un compilatore C++) che è un particolare software che a partire da un programma scritto in un determinato linguaggio di programmazione (il *programma sorgente* C++) può generare un *programma oggetto eseguibile*. Tutto ciò è ottenuto dal compilatore traducendo in codice binario comprensibile dalla macchina le istruzioni contenute nel sorgente. Tale processo, che può variare e prevedere diverse fasi in dipendenza del compilatore stesso e del suo funzionamento, richiede dal programmatore l'esecuzione di alcune operazioni che possono essere ricondotte in breve alle seguenti:

- Scrittura del sorgente e salvataggio su disco del sorgente stesso. Tale operazione viene effettuata con l'ausilio di un *editor*: software che permette appunto di inserire il testo con le istruzioni del programma e salvarlo su memoria di massa. In questa fase il programma è semplicemente una sequenza di caratteri priva di significato per il computer e, quindi, può essere scritto utilizzando un qualsiasi software che generi *testo puro*. Non va bene, in genere, un programma di elaborazione testi perché verrebbero inseriti nel testo anche i caratteri di formattazione (grassetto, corsivo, dimensioni caratteri, colori, ecc...) a meno che il software non preveda (e in genere è possibile) la generazione di testo puro. Conviene però utilizzare un editor poiché si tratta di uno strumento creato per chi sviluppa software e quindi provvisto di facilitazioni e aiuti per i programmatori.
- Compilazione del sorgente. Tale fase richiede appunto un *compilatore* che è in grado di tradurre il sorgente in codici eseguibili *per una determinata CPU* e *per un determinato Sistema Operativo*. Se il compilatore non è in condizione di generare il codice eseguibile perché è stato commesso qualche errore di sintassi, viene generata la lista degli errori. In questo caso, anche se qualche compilatore produce il codice eseguibile, non se ne può tenere conto avendo, questo, comportamenti imprevedibili. Bisogna riprendere l'editing del sorgente, correggere le istruzioni che hanno generato errori e ripetere la compilazione con il

programma modificato.

- Esecuzione e test del programma. Una volta generato il codice eseguibile è necessario verificare se il programma fornisce i risultati attesi. Scelti i dati sui quali effettuare i test, si lancia l'esecuzione del programma fornendo tutti i dati previsti dai test. Se i risultati prodotti sono quelli attesi: bene! In genere il caso più frequente è quello in cui si verificano situazioni anomale: risultati non coerenti rispetto alle attese, blocchi del programma. Si rende necessario effettuare un monitoraggio dell'esecuzione del programma per verificare quali sono le situazioni che non permettono al programma stesso di funzionare nella modalità corretta. In questi casi è necessario utilizzare un *debugger* che è un software che permette, per esempio, di controllare come variano, nel corso dell'esecuzione, i valori contenuti nelle variabili utilizzate dal programma o fare eseguire le singole istruzioni verificandone gli effetti.

In definitiva, da quanto esposto, per completare il processo di compilazione per la produzione del codice eseguibile, è necessario utilizzare una serie di strumenti software. Quanto meno bisogna utilizzare: editor, compilatore, debugger. In alcuni casi, per completare l'opera del compilatore è necessario un *linker*, in altri casi, in dipendenza del tipo di compilatore, occorrono anche altri strumenti software, per esempio per l'esecuzione del programma compilato. Se gli strumenti riescono ad integrarsi fra di loro consentendo, per esempio, all'editor di dialogare con il compilatore, si può più rapidamente cercare e correggere la riga del sorgente dove è stato commesso un errore. Gli strumenti del progetto GNU hanno questa capacità di integrazione: Emacs è fornito di librerie che permettono di interagire in maniera proficua con GCC e GDB, consentendo una maggiore efficienza e velocità, da parte del programmatore, nel processo di sviluppo.

Terminologia di *GNU Emacs*

La prima fase dello sviluppo di un programma prevede la scrittura del sorgente con l'utilizzo di un editor.

GNU Emacs è uno strumento estremamente potente (e complesso) che, a definirlo solo un editor, sicuramente si pecca di approssimazione. Tanto per avere un idea si tenga presente che il manuale d'uso, disponibile presso il sito della FSF, è composto da più di 600 pagine. In realtà può essere utilizzato come un ambiente completo di sviluppo, come uno strumento per la navigazione all'interno dell'albero delle directory, come interfaccia verso il sistema operativo, come agenda, come piattaforma per giochi, client di posta elettronica ecc.. e inoltre è completamente personalizzabile ed espandibile: basta conoscere e sapere utilizzare il linguaggio con cui è sviluppato. In definitiva si tratta di un ambiente per l'editing e l'esecuzione di macroistruzioni in Lisp (il nome deriva da Editor MACroS). Dalla versione 24 è integrato in Emacs un gestore di pacchetti aggiuntivi riguardanti applicazioni diverse che ne espandono le potenzialità. In questi appunti si tratteranno le caratteristiche di Emacs derivanti dall'installazione dei pacchetti ECB

(Emacs Code Browser), CEDET (Collection of Emacs Development Tools), YASnippet, dalla configurazione suggerita dagli utilizzatori in rete (vedere Appendice 2 per maggiori dettagli) e da altre configurazioni. In Appendice è riportato pure il link da cui scaricare la configurazione, pronta per l'uso, cui si fa riferimento in questi appunti.

Tutta questa potenza, naturalmente, ha un prezzo. Da molti è considerato un prodotto *ostico* da imparare ad usare ed effettivamente anche la terminologia adottata non ne agevola l'utilizzo, specie per chi lo usa per le prime volte. Occorre considerare che si tratta di un prodotto nato in ambienti non grafici (nel 1974 Richard Stallman aggiunge il supporto delle Macro a un progetto del 1972, iniziando la storia di Emacs come lo si conosce oggi) e che conserva alcuni modi di operare di quegli ambienti. Per poterlo utilizzare proficuamente è necessario entrare nella sua *filosofia*.

Inizialmente l'interazione con Emacs, alla stessa stregua di tutti i programmi nati in epoca dove ancora le interfacce grafiche erano inesistenti o poco diffuse, avveniva, e tuttora può avvenire, con comandi che vengono impartiti utilizzando delle combinazioni di caratteri o comandi digitati all'interno di un apposito ambiente. Le versioni attuali di Emacs permettono una interazione, più in linea con le direttive delle moderne interfacce grafiche, per mezzo dell'uso del mouse e di pulsanti e non dissimile rispetto ad altri software con caratteristiche e potenzialità paragonabili. Le combinazioni di caratteri possono essere utilizzate come *scorciatoie* dei comandi. L'interazione con l'ambiente è possibile, come comune, anche tramite pulsanti associati a determinate azioni o menù da cui si possono selezionare le operazioni desiderate.

In generale i caratteri associati ai comandi sono accompagnati da tasti modificatori:

Control *(indicato con* `C`*),* ***Meta*** *(indicato con* `M`*)*: il primo è il tasto <CTRL> che si trova in doppia copia in tutte le tastiere moderne a lato della barra spaziatrice, il secondo è il tasto <ALT> a sinistra della barra spaziatrice. Il nome *Meta* deriva dalla stampigliatura presente nei tasti nei sistemi Unix. Nelle tastiere oggi frequentemente utilizzate la stampigliatura è stata sostituita, appunto, da *Alt*. Questi tasti (chiamati *modificatori* perché modificano l'effetto del tasto cui sono accoppiati) vanno usati insieme con un altro tasto. Per esempio, se si legge la combinazione `M-x` (per richiamare la riga di comando del Minibuffer), vuol dire premere il tasto <ALT> e, senza rilasciarlo, premere il tasto *x* e quindi rilasciare i due tasti. Il tasto <ALT> può essere sostituito dal tasto <ESC> presente nelle tastiere in alto a sinistra. Nell'esempio di prima `M-x` può essere eseguito, in alternativa a quanto esposto prima, digitando <ESC>, rilasciandolo e digitando poi *x*.

Sono usati anche altri termini che è bene chiarire:

- ***Buffer***: si tratta della zona di memoria centrale che contiene il testo che si sta editando. Un file di testo che si vuole editare viene caricato in un buffer. Se poi si vuole salvare su disco il testo visualizzato nell'area di lavoro corrente (quella dove si trova il cursore di inserimento testo), si effettuerà la selezione, per esempio, dal menù *File* > *Save* o si utilizzerà l'apposito pulsante.
- ***Window***: si tratta della parte di schermo in cui, per esempio, è visualizzato un buffer e si può interagire con esso e quindi esiste, per esempio, un comando per dividere la finestra (*File* > *New Window on Right* o *New Window Below*) e in questo modo l'area di lavoro viene suddivisa in due parti in modo da permettere l'editing di un file diverso in ogni window. *File* > *Remove Other Windows* espande l'area di lavoro corrente in modo da coprire l'intera area disponibile.
- ***Frame***: si tratta in questo caso di tutta la zona dello schermo che contiene sia l'applicazione

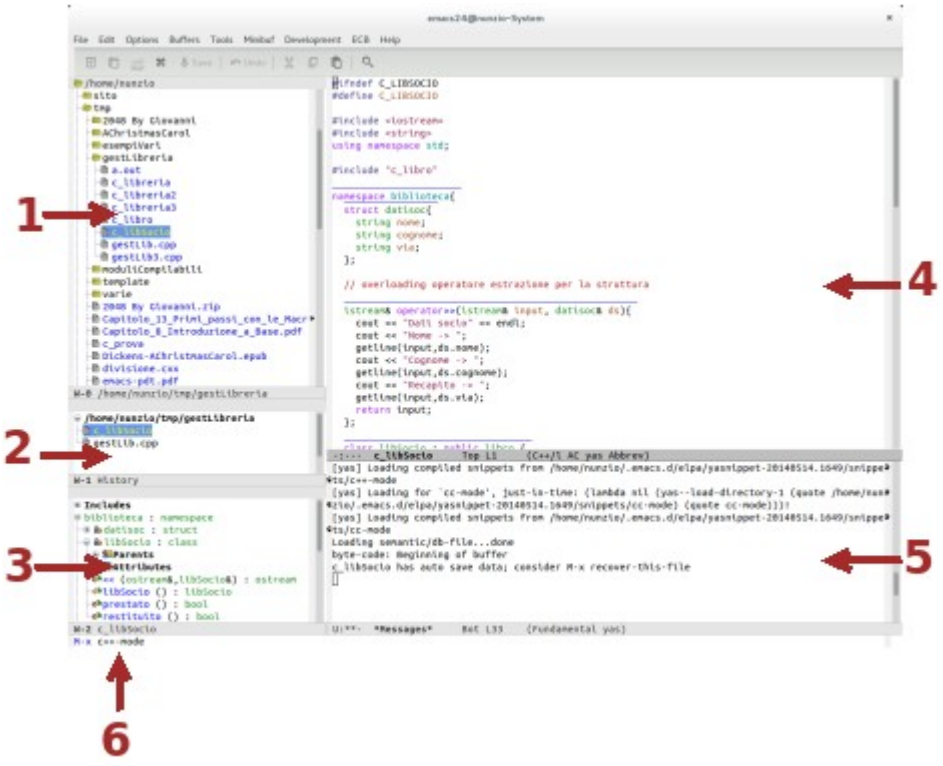
sia il testo, quella che negli ambienti desktop è definita finestra. Se si esegue, per esempio *File > New Frame*, ci si ritroverà con un duplicato di tutta l'applicazione. In realtà non si tratta di una nuova istanza di Emacs poiché se, per esempio, in un frame si apre un nuovo testo, lo si ritroverà anche nell'altro. Quindi in definitiva si tratta di due visioni separate dello stesso ambiente

Emacs, quando gira in un ambiente grafico, fa uso dei pulsanti del mouse. Anche in questo caso è bene chiarire la terminologia utilizzata:

Mouse-1, ***Mouse-2***, ***Mouse-3***: si tratta rispettivamente della pressione dei pulsanti sinistro, centrale, destro del mouse. Nei mouse forniti di soli due tasti, il pulsante centrale è emulato dalla pressione contemporanea dei due tasti presenti. La pressione di uno dei tasti del mouse può essere anche accompagnata dalla pressione dei tasti modificatori. Per esempio `C-Mouse-1` si esegue tenendo premuto il tasto <CTRL> e premendo il tasto sinistro del mouse (in Emacs mostra l'elenco completo dei buffer). `Mouse-2` è in genere usato per avviare qualcosa (equivale al doppio clic sinistro in altri ambienti): per esempio sul nome di un file nella finestra di navigazione apre il file nella finestra dell'editor. Può essere mentalmente associato a un comando del tipo: *Vai a ...*

GNU Emacs: interfaccia e orientamento

L'avvio di default di Emacs, nella configurazione di riferimento di questi appunti, visualizza l'area di lavoro suddivisa in 6 parti:



Le tre finestre sulla sinistra fanno parte della installazione del pacchetto ECB e permettono una facile gestione dei sorgenti, specie se si deve lavorare in contemporanea con più di uno, con variabili strutturate, con classi ed oggetti: nella **1** è visualizzato l'albero delle directory a partire dalla home dell'utente. `Mouse-2` apre una directory o carica il file nell'editor.

La **2** (History) mostra lo storico dei file che sono stati aperti nell'editor: `Mouse-2` attiva il file nella finestra dell'editor. È bene tenere presente che la finestra mostra i file aperti per l'editing e non *tutti* i file aperti: mancano dall'elenco il buffer di avvio (quello con il logo), il buffer ***scratch*** (utilizzabile come post-it per scrivere brevi appunti che non saranno conservati), il buffer con i messaggi (visualizzato nella **5**), buffer con messaggi di compilazione, di debugging ecc.. (l'elenco completo, come già in precedenza ricordato, si può ottenere con `C-Mouse-1` nella finestra dell'editor o in quella dei messaggi o, direttamente, dal menù *Buffer*). Nella finestra sono elencati anche i file che, aperti in precedenza, sono stati chiusi (i nomi sono visualizzati in grigio chiaro) e per fare visualizzare solamente i file attualmente presenti in buffer di editing basta selezionare *ECB > Add all buffers to history*.

La **3** è dedicata alla visualizzazione dell'elenco dei simboli utilizzati nel sorgente in editing: funzioni, prototipi, classi e metodi, strutture e membri. `Mouse-2` su un simbolo posiziona il cursore, nella finestra di editing (**4**), nel punto in cui si trova il codice corrispondente.

la finestra **4** è dedicata all'editing del documento evidenziato nella **1** e nella **2**.

La **5** è dedicata alla visualizzazione dei messaggi che l'ambiente comunica: output della compilazione, output dell'esecuzione di un programma ecc... Quando non è attivata automaticamente come all'avvio di Emacs, durante la compilazione o l'esecuzione di un programma, la visualizzazione può essere abilitata agendo dal menù *ECB > Toggle compile window* che funziona da interruttore per visualizzare/nascondere la finestra. Quando la **5** non è visualizzata, la **4** si espande occupando lo spazio disponibile.

La **6** mostra il ***Minibuffer***: la riga di comando per interagire con alcune funzionalità dell'ambiente. Originariamente unica interfaccia verso i comandi. `M-x` (o in alternativa la pressione del tasto <ESC> seguito dal tasto x) porta il cursore nella riga del Minibuffer pronto per accettare un comando dell'ambiente. La tripla pressione consecutiva del tasto <ESC> (<ESC> <ESC> <ESC> in successione) esce dall'ambiente di introduzione di comandi e riporta il cursore nella posizione, e nella finestra, in cui si trovava prima che fosse stato richiamato l'inserimento nel Minibuffer.

L'interruttore del menù *ECB > Toggle visibility of ECB windows* nasconde/visualizza le finestre **1**, **2** e **3** e, qualora queste siano nascoste, il frame è ricoperto totalmente da **4** e **5**. Il Minibuffer è sempre accessibile.

Quando si comincia ad editare un file il punto di inserimento viene mostrato come un rettangolino scuro e tutto ciò che è digitato da tastiera viene inserito nella posizione del punto di inserimento, eventuale testo già esistente nella posizione del punto di inserimento o alla sua destra, viene spostato per lasciare posto ai nuovi caratteri digitati.

Per selezionare una parte di testo (*Region* nel linguaggio utilizzato da Emacs) si porta il punto di inserimento nel primo carattere della parte di testo interessata (basta spostare il mouse nella posizione desiderata e premere il pulsante sinistro `Mouse-1`), e, tenendo premuto il pulsante, si trascina il mouse stesso fino alla posizione immediatamente successiva a quella finale. In alternativa, una volta portato il punto di inserimento nel primo carattere del testo da selezionare, si può premere `Mouse-3` nel punto finale della selezione che si vuole effettuare.

- Per cancellare il testo selezionato basta premere il tasto *Canc*
- Per spostare il testo in una nuova posizione si può, come solito, premere il tasto *Taglia*, spostare il punto di inserimento nella nuova posizione e premere il tasto *Incolla*. Un modo

ancora più rapido è quello di sfruttare le caratteristiche di *Taglia*, *Copia*, *Incolla* dei tasti del mouse. Quando si è selezionato il testo si è già effettuata l'operazione di Copia. Se si vuole spostare il testo basta premere <CANC>: il testo a questo punto scompare. Si sposta il mouse nella posizione del nuovo inserimento e si preme `Mouse-2`. Selezionando *Edit* > *Paste from kill menu*, si può scegliere, fra tutte le selezioni cancellate nella sessione di lavoro, quale incollare nel punto di inserimento attuale.

- Il testo selezionato può essere mandato in stampa scegliendo *File* > *Print Region*. In alternativa si può mandare in stampa direttamente la parte di testo utilizzando i comandi riconosciuti da Emacs: nel caso specifico `M-x` (fa passare al Minibuffer) e quindi digitando `print-region`.

Per mandare in stampa l'intero file si può agire da *File* > *Print Buffer*.

Editing di un sorgente C++

Essendo Emacs concepito come editor e strumento di lavoro per programmatori, mette a disposizione tutta una serie di facilitazioni, anche espandibili in seguito all'installazione di pacchetti aggiuntivi di cui si è già parlato, utili quando il file che si edita è un sorgente di un programma scritto in un determinato linguaggio di programmazione. Dette facilitazioni, per il C++, sono rese disponibili quando Emacs *passa in modalità C++*, ovvero si predispone per considerare che il file di testo su cui si lavora è un sorgente C++.

Emacs può passare in modalità C++ in due modi:

1. ***Riconoscimento automatico dell'estensione***: quando ad un file viene assegnato un nome che termina con una delle estensioni (l'ultima parte conclusiva del nome assegnato al file) standard associate a un sorgente C++: `.cxx` (quella utilizzata in questi appunti), `.cpp`, `.cc` o `.C`.
2. ***Richiamo esplicito della modalità***: si attiva richiamando il Minibuffer (con la combinazione `M-x`) e scrivendo il comando:

   ```
   c++-mode
   ```

   modalità utile quando, per esempio, si deve editare un file di intestazione o di definizione di una classe che non ha una estensione fra quelle riconosciute.

Qualunque sia il sistema con cui viene attivata la modalità, questa comporta, come primo effetto visivo, la modifica delle voci presenti nella barra dei menù evidenziata dalla comparsa dell'opzione *C++*.

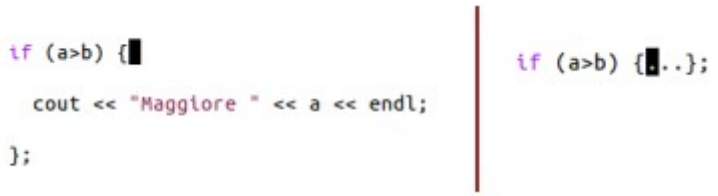
Indentazione automatica: quando si inserisce la riga successiva, rispetto a quella in cui è presente una parentesi di inizio blocco (`{`), questa viene rientrata verso destra non appena Emacs si rende conto che la riga è terminata (in genere `;`). Se si inserisce una parentesi di fine blocco (`}`) la riga viene spostata verso sinistra. Le rientranze vengono gestite in modo automatico anche quando, per esempio, l'unica istruzione in una `if` termina la struttura di controllo. L'indentazione è una facilitazione importante per il programmatore e il fatto che Emacs la metta a disposizione, gestendola in modo automatico, può mettere al riparo da alcuni errori che frequentemente vengono commessi come: dimenticare qualche parantesi o il carattere di fine linea. Se l'indentazione non è più corrispondente per, ad esempio, la cancellazione di una struttura, il tasto <F9> reindenta, in modo corretto, l'intero file (equivale a scegliere *C++* > *Indent Line or Region* ma senza necessità di

definire un blocco di linee). La singola riga può essere indentata premendo, in qualsiasi punto della riga, <TAB>.

Colorazione sintassi e Decorate Tags: le parole chiavi sono colorate in un modo, i commenti in un altro, le stringhe in un altro ancora e così via. In tal modo ci si può più agevolmente accorgere della mancanza di un delimitatore o degli errori nella trascrizione di una parola chiave. Inoltre le funzioni sono precedute da una linea colorata che permette di rintracciarle agevolmente all'interno del sorgente.

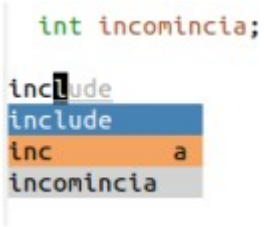
Abbinamento di parentesi: anche questa è una agevolazione che permette di evitare errori, molto frequenti e insidiosi, dovuti all'omissione di qualche parentesi chiusa. Non appena si sposta il cursore dopo l'immissione di una parentesi chiusa, viene evidenziata la coppia di parentesi che racchiude il blocco e il colore è diverso se c'è corrispondenza fra parentesi (si sta chiudendo una parantesi di un tipo e l'ultima aperta era dello stesso tipo) o non c'è corrispondenza (la parentesi chiusa non è dello stesso tipo dell'ultima aperta).

Contrazione/Espansione blocchi:



la combinazione `C-+` (tenere premuto <CTRL> e senza rilasciarlo premere il tasto con il simbolo + ma non quello del tastierino numerico) in una riga qualsiasi, è un interruttore che riduce il blocco di codice contenete il cursore, ad una sola riga, nascondendo il codice interno, o ne riabilita la visualizzazione. Si tratta di una facilitazione che consente di seguire più facilmente la logica della sequenza dell'algoritmo utilizzato, evitando la visualizzazione di dettagli che potrebbero distrarre l'attenzione.

Autocompletamento:



funzione che permette di ridurre la quantità di caratteri da digitare. Nell'esempio dopo aver digitato `inc`, viene visualizzato un menù con i possibili completamenti. In questo caso si può scegliere, dall'alto in basso, fra una parola chiave uno *snippet* o una variabile. Con le frecce direzionali <SU><GIU>, si può cambiare la scelta. Il tasto <TAB> conferma la scelta o si può continuare a digitare qualsiasi altra cosa.

Snippet o frammenti di codice:

si tratta di frammenti di codice riproducenti le strutture fondamentali del linguaggio C++ e richiamati utilizzando opportune chiavi. Per esempio se si digita `if` seguito da <TAB> viene inserito lo scheletro della struttura e il cursore viene portato nella condizione. Dopo averla specificata, un nuovo <TAB> porta il cursore di inserimento dentro le parentesi e si è pronti per l'inserimento delle opportune istruzioni. Per strutture più complesse <TAB> sposta da una zona all'altra e finché ci saranno parti evidenziate. Esistono snippet per tutte le strutture previste dal linguaggio. Quando potrebbe presentarsi qualche ambiguità, cioè alla stessa chiave corrispondono più snippet, (nell'immagine il caso della digitazione del carattere `f` seguito da <TAB>), dopo la digitazione di <INVIO> viene presentato un menù da cui poter scegliere lo snippet più opportuno. Gli snippet possono essere inseriti anche selezionandoli dal menù *YASnippet* > *cc-mode* o *c++-mode* ma il sistema più rapido e più efficiente è quello di usare le parole chiavi. L'uso degli snippet fornisce un doppio vantaggio durante lo sviluppo dei programmi: velocità di inserimento dovuta alla digitazione di pochi caratteri al posto di strutture talvolta lunghe, sicurezza di non commettere errori di digitazione o di errato utilizzo di strutture.

L'elenco degli snippet definiti nella configurazione di riferimento è riportato nell'Appendice 1 di questi appunti.

Suggerimento semantico:

la combinazione di tasti <CTRL><INVIO> visualizza, nella posizione in cui si trova il puntatore del mouse, un tooltip con l'elenco dei possibili inserimenti validi per la variabile o oggetto. Utilizzabile se, come nell'esempio, si tratta di una variabile strutturata per la scelta del membro o per la scelta del metodo di un oggetto. Vale anche se la struttura o la classe sono definiti in file a parte inclusi, con una direttiva del tipo `include`, nel file attuale. Ogni volta che nell'editor si cambia il buffer visualizzato, il modulo Semantic di Emacs effettua il ***parsing*** (analisi grammaticale) del file inattivo se modificato, conserva le informazioni in un database e quindi è in grado di fornire i suggerimenti coerenti con il contesto.

Compilazione ed esecuzione

Il compilatore C++ presente in Linux è, come accennato in precedenza, contenuto nella suite GCC. Si avvia normalmente con un comando dalla shell, ma si può avviare la compilazione di un programma all'interno di Emacs stesso.

È conveniente selezionare *Tools* > *Compile* e quindi inserire la riga di comando dal *Minibuffer*. Se il file sorgente ha il nome `prova.cxx` e si vuole che il file oggetto si chiami `prova`, la riga di comando da digitare nel *Minibuffer* è:

```
c++ -o prova prova.cxx
```

oppure:

```
c++ -std=c++11 -o prova prova.cxx
```

dove:

- `c++` è il nome dell'eseguibile per la compilazione di programmi C++.
- L'opzione `-std=c++11` è necessario aggiungerla solamente quando nel sorgente sono specificate caratteristiche implementate nello standard C++11 e non presenti nelle versioni precedenti.
- L'opzione `-o` serve per specificare il nome che dovrà avere il file oggetto. Se l'opzione non è presente il nome del file oggetto sarà `a.out`.
- Il nome del file sorgente chiude la linea.

Confermata la riga si avvia la compilazione e se questa va a buon fine, con il messaggio `Compilation finished`, il file oggetto è stato creato correttamente nella stessa directory in cui si trova il sorgente.

Se la compilazione non è andata a buon fine, nella window dedicata ai messaggi del compilatore, viene visualizzato l'elenco dei messaggi di errore con indicazioni sulla linea del sorgente. In questo caso il messaggio visualizzato è del tipo `Compilation exited abnormally`.

```
#include <iostream>
using namespace std;

int main()
{
  int a, b, c;

  // input da elaborare

  cout << "Primo programma in C++" << endl;

  cout << "primo numero ";
  cin  >> a;
  cout << "secondo numero ";
  cin  >> b;

  // elaborazione e output

  cic = a+b;
  cout << "risultato somma " << c << endl;
:---  prova.cxx      Top L19    (C++/l Abbrev)----------------
-*- mode: compilation; default-directory: "/home/ubuntu/" -*-
Compilation started at Thu Jan  2 12:33:40

c++ prova.cxx
prova.cxx: In function 'int main()':
prova.cxx:19:3: error: 'cic' was not declared in this scope

Compilation exited abnormally with code 1 at Thu Jan  2 12:33
```

Nell'esempio viene evidenziato che alla linea 19 è usata una variabile non dichiarata, infatti nella penultima riga visualizzata del sorgente è usata la variabile `cic` non dichiarata.

Non è necessario sapere quale è la linea 19 per andare a correggere l'errore (anche se il numero di riga in cui è presente il cursore, è riportato nella riga di stato della finestra dell'editor), basta, nella finestra dei messaggi con i risultati della compilazione, premere `Mouse-2` (*Vai a...*) nella riga dell'errore e ci si sposta rapidamente nelle riga specificata nelle finestra dell'editor.

Corretti tutti gli errori rilevati dal compilatore e salvata su disco la nuova versione del sorgente, si può procedere con una nuova compilazione.

Una volta che la compilazione del programma va a buon fine, bisogna lanciarne l'esecuzione per provare se i risultati forniti coincidono con quelli attesi. Naturalmente il lancio dell'eseguibile può essere effettuato da una shell avviata da fuori come programma esterno, ma può risultare comodo effettuare tale operazione direttamente da Emacs.

Portandosi sulla riga del *Minibuffer* con `M-x` e digitando `eshell`, si richiede di lanciare la shell testuale nella finestra dei *Messaggi*. Si tratta di una shell che funziona esattamente come qualunque altra shell avviata esternamente. Il tasto <SU> permette di scorrere le righe inserite in precedenza (storico dei comandi), <TAB> permette il completamento della linea quando possibile. Da qui si può, inoltre, lanciare l'oggetto eseguibile e verificarne il funzionamento.

Il debugging dei programmi

Un programma durante l'intera fase del suo sviluppo, dal sorgente alla compilazione e generazione del file oggetto, al testing, può evidenziare errori e malfunzionamenti che possono essere, orientativamente, ricondotti a due categorie:

- ***Errori rilevati in fase di compilazione***: si tratta degli errori rilevati dal compilatore e

derivati da un uso errato delle parole chiavi o delle strutture sintattiche del linguaggio utilizzato per la scrittura del sorgente. Il compilatore, per vari motivi, non è in condizioni di generare il codice oggetto. Sono questi, tutto sommato, gli errori meno preoccupanti perché semplici da rintracciare e correggere: il compilatore fornisce una indicazione di massima sul tipo di errore commesso e sulla localizzazione della riga del sorgente dove è stato rilevato. Il messaggio di errore fornito può essere ambiguo e la riga indicata può essere anche la successiva a quella nella quale è localizzato effettivamente l'errore. Il compilatore genera l'errore quando non sa come tradurre una istruzione e questo può avvenire anche dopo la linea non corretta: in seguito all'errata interpretazione ci si aspetta qualcosa che, in effetti, non c'è e quindi l'errore viene individuato in quello che manca quando invece è quello che c'è che non va. In ogni caso, presa confidenza con i messaggi generati dal compilatore, è abbastanza agevole, in generale, risolvere l'errore.

- ***Errori rilevati in fase di run-time***: sono gli errori che si evidenziano quando si esegue il programma. I risultati prodotti non sono quelli attesi. Sono gli errori più insidiosi perché spesso difficili da rintracciare principalmente perché le uniche cose che il programmatore può controllare sono gli output, ma in realtà si tratta solamente dell'ultimo anello della catena: gli output sono semplicemente il risultato, visualizzato su schermo per esempio, di calcoli, che evidentemente sono errati in qualche punto, ma che non sono immediatamente controllabili.

Quello che è necessario per la correzione degli errori di run-time è avere a disposizione strumenti per il monitoraggio dell'esecuzione del programma, qualcosa che permetta, per esempio, di *vedere dentro il computer* gli effetti dell'esecuzione delle istruzioni. I debugger sono programmi che forniscono strumenti per il controllo dell'esecuzione di un programma. GDB è il debugger del progetto GNU di cui verranno esaminati i comandi di uso più comune e la sua interazione con l'ambiente Emacs.

Prima di entrare nello specifico di una *sessione di debugging*, è bene chiarire alcuni termini di uso comune.

- ***Breakpoint***: nelle sessioni di debugging si indica con questo termine il punto in cui si deve bloccare l'esecuzione del programma. Settare il breakpoint su una riga del programma comporta il blocco dell'esecuzione del programma immediatamente prima dell'esecuzione della riga.
- ***Watchpoint***: punti di osservazione delle variabili. Settare il watchpoint su una variabile o una espressione comporta il ricevere informazioni su come e quando varia il contenuto di una variabile o il valore dell'espressione.
- ***Step***: si tratta in questo caso di un modo di eseguire il programma un passo per volta. Ogni volta che vengono eseguite una o più istruzioni, a seconda del comando impartito al debugger, l'esecuzione del programma si blocca permettendo così di controllare le modifiche che la stessa ha apportato, per esempio, nei valori contenuti in una variabile.

Il supporto di Emacs al debugging

In Emacs sono incluse librerie che ne permettono una integrazione con alcuni debugger e, in modo particolare, con GDB, il debugger del progetto GNU.

Per prima cosa se si vuole effettuare il debugging di un programma è necessario compilarlo con una opzione che permette di inserire nell'oggetto le informazioni utili per il debugging. Tali informazioni, infatti, poiché occupano spazio facendo aumentare le dimensioni del file oggetto, di solito non vengono incluse. Per aggiungere tali informazioni bisogna utilizzare il flag `-g` nella riga di comando della compilazione.

In definitiva, per la compilazione del file sorgente `prova.cxx` comprendente le informazioni per il debugging, la linea di comando da digitare, da Emacs selezionando *Tools* > *Compile*, è:

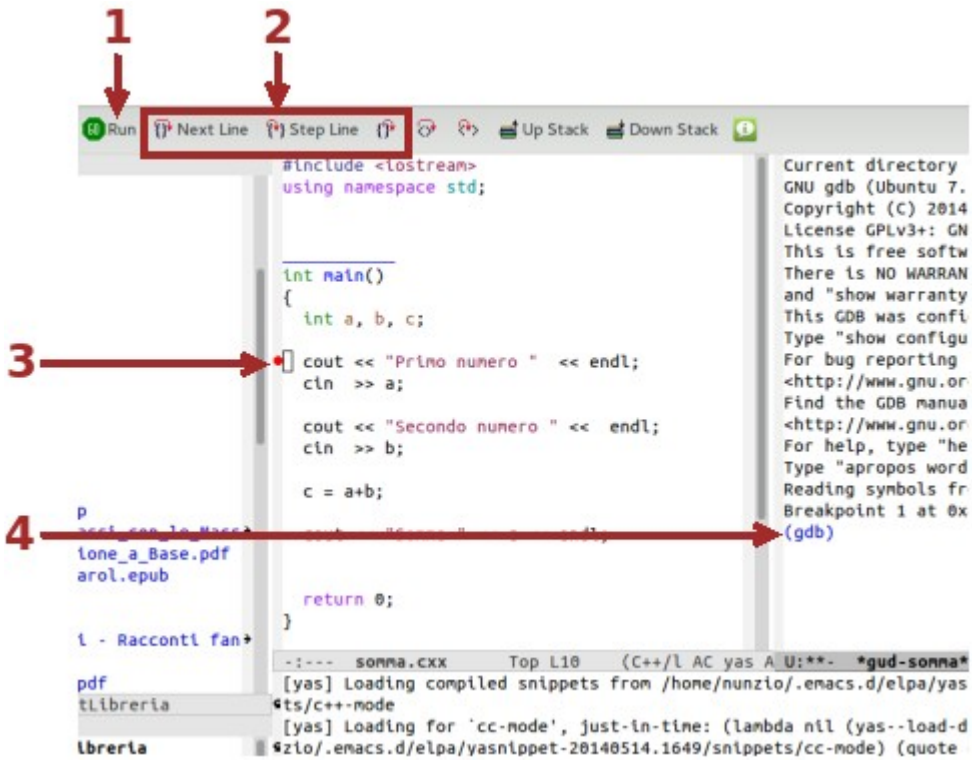
```
c++ -g -o somma somma.cxx
```

In questo caso viene generato l'oggetto `somma` contenete le informazioni che servono a GDB. L'integrazione del debugger in Emacs permette l'interazione con il sorgente del programma sul quale è stata avviata la sessione di debugging.

Per avviare la sessione di debugging è conveniente innanzi tutto dividere la window dell'editor in due parti (cursore nella finestra dell'editor e poi selezionare *File* > *New Window on Right*) e, successivamente, selezionare *Tools* > *Debugger (GDB)*. Nel minibuffer viene proposto il comando per l'avvio:

```
gdb -i=mi somma
```

Confermata la riga di comando e avviato il debugger, cambia la barra delle icone di avvio rapido. Per la descrizione delle sezioni di debugging di cui si tratterà, i pulsanti più importanti sono:



Il pulsante **1** avvia o continua l'esecuzione del programma fino al prossimo breakpoint, se esiste, o fino all'ultima istruzione.

Il gruppo **2** controlla l'esecuzione del programma in modalità step-by-step. Selezionando la prima icona (*Next Line*) si permette l'esecuzione della sola prossima istruzione del programma. Se

l'istruzione è una chiamata di funzione viene trattata come unica istruzione e le istruzioni contenute nella funzione vengono eseguite senza interruzioni. La seconda icona (*Step Line*) consente invece di *entrare* nella funzione. La terza icona (*Finish Function*) consente, quando si sta eseguendo una istruzione contenuta in una funzione, di eseguire le rimanenti istruzioni della funzione senza interruzioni.

I breakpoint si possono settare con `Mouse-1` all'inizio della riga del sorgente. Un punto rosso (**3**) indica il punto di arresto che può essere tolto premendo nuovamente `Mouse-1` sul punto.

L'avvio del debugger provoca anche la visualizzazione del nuovo menù *Gud* le cui opzioni consentono l'esecuzione controllata del programma. Il sotto menù *Gud* > *GDB-Windows* > *Locals*, per esempio, permette la visualizzazione delle variabili locali del programma in esecuzione (watchpoint delle variabili locali), in modo da osservare, durante l'esecuzione delle istruzioni del programma come cambia il contenuto delle variabili osservate e rendersi conto, per esempio, quale è l'istruzione che produce un risultato non coerente con quello atteso.

Comandi dal prompt di GDB

Si può interagire con GDB anche utilizzando comandi impartiti dal prompt (**4**) del debugger. Di seguito è riportata una tabella con i comandi di uso più frequente.

Comando	***Significato ed esempi d'uso***
`run` abbreviabile con `r`	Lancia l'esecuzione del programma specificato all'avvio. Se non ci sono breakpoint l'esecuzione termina con la fine del programma, altrimenti viene bloccata al primo breakpoint
`print` abbreviabile con `p`	Visualizza il valore della variabile o dell'espressione specificate: 1. `print beta` (mostra l'***attuale*** valore contenuto nella variabile specificata) 2. `print beta+10` (mostra il valore dell'espressione. Il valore attuale della variabile viene aumentato di `10`) 3. `pvector numeri` (mostra il contenuto del vettore. Esiste l'equivalente per tutti gli oggetti delle STL. Per es: `plist`, `pmap` ... Questa possibilità è resa disponibile dalla configurazione di GDB compresa nella configurazione di riferimento di questi appunti)
`continue` abbreviabile con `c`	Continua l'esecuzione del programma dopo un breakpoint: 1. `continue` (continua l'esecuzione del programma a partire dall'istruzione contenuta nella riga del breakpoint e fino al prossimo break o alla fine del programma)
`next` abbreviabile con `n`	Esegue la prossima linea di programma. Se l'istruzione è una chiamata di funzione, viene eseguita come singola istruzione. Funzionalità utile se non si vuole entrare nella funzione e quando si tratta di chiamate a funzioni di libreria.
`step` abbreviabile con `s`	Esegue la prossima istruzione. Se l'istruzione è una chiamata ad una funzione, viene eseguita la prima istruzione della funzione.
`quit` abbreviabile con `q`	Permette l'uscita da GDB e fa terminare la sessione di debugging.

Osservare l'esecuzione di un programma

Per trovare le istruzioni di un programma che non permettono di ottenere i risultati attesi può essere di aiuto esaminare, istruzione per istruzione, le modifiche apportate nelle variabili: esecuzione step-by-step del programma.

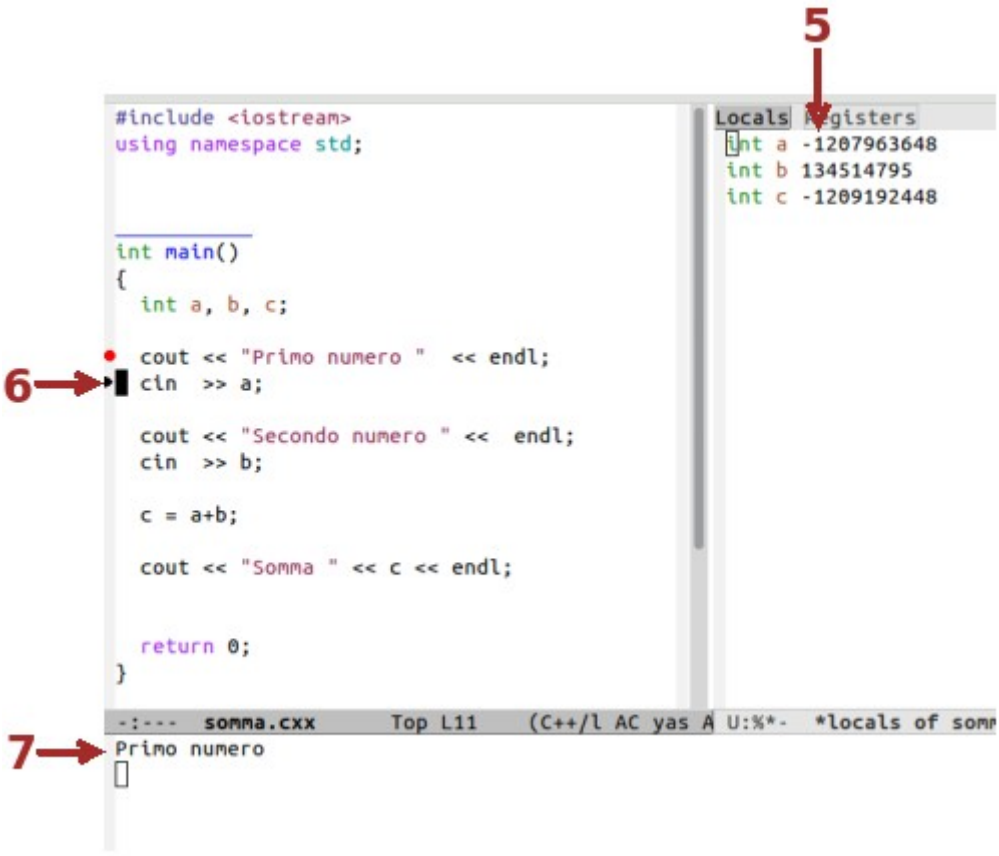
- Selezionare *Tools* > *Compile...* e compilare il programma con il flag per l'inclusione delle informazioni per il debugging.

  ```
  c++ -g -o somma somma.cxx
  ```

- Selezionare *File* > *New Window on Right* (oppure direttamente nella finestra dell'editor si digita la combinazione di tasti `C-x 3`) in modo da avere l'area di lavoro divisa in due parti.
- Dal menù *Tools* si seleziona *Debugger (GDB)...* per l'avvio della sessione di debugging.

  ```
  gdb -i=mi somma
  ```

- Nella prima istruzione del programma, subito dopo la dichiarazione delle variabile, si inserisce un breakpoint. Basta spostarsi sul bordo sinistro della riga che contiene l'istruzione e premere `Mouse-1`.
- Si seleziona *Gud* > *GDB-Windows* > *Locals* per mettere in una delle window su cui è stata suddivisa l'area dell'editor, le variabili locali e i valori in esse contenute. La finestra si aggiorna automaticamente ad ogni modifica del valore di una variabile.
- Si preme il pulsante *GO* (**1**) per avviare l'esecuzione del programma che si blocca prima dell'istruzione sulla quale è stato impostato il breakpoint.



- Inizialmente, all'avvio, le variabili dichiarate prendono valori casuali (**5**).
- Selezionando l'icona *Next Line* si esegue la prossima istruzione. Una freccia nera sul bordo sinistro della riga del sorgente, indica l'istruzione che verrà eseguita nel prossimo step (**6**).
- Nella finestra dei messaggi (**7**) si può visionare l'output del programma e si può interagire con esso, per esempio, introducendo gli input richiesti.

Se si ha necessità di inserire comandi dal prompt `(gdb)` si può posizionare il cursore nella finestra dei messaggi/esecuzione del programma (**7**), visualizzare l'elenco dei buffer (`C-Mouse-1`) e scegliere l'ambiente interattivo del debugger `gud-somma` (nel caso in esame in cui l'oggetto si chiama `somma`). Al posto dell'output del programma viene visualizzato il prompt di GDB e si possono inserire i comandi. Per ritornare all'output del programma, per esempio, per passare all'esecuzione della successiva istruzione, visualizzare nuovamente l'elenco dei buffer (`C-Mouse-1`) e selezionare `input/output somma`.

Se si è sicuri del corretto funzionamento di una parte del codice, si possono impostare diversi breakpoint nei punti critici ed eseguire più velocemente le istruzioni intermedie premendo GO/run (**1**) dopo ogni breakpoint. Anche l'impostazione di un watchpoint su una variabile (`Mouse-1` sul nome della variabile) blocca l'esecuzione del programma ad ogni modifica del valore contenuto nella variabile ma esegue le istruzioni che non la modificano senza interruzioni.

Appendice 1: snippet

Gli snippet sono frammenti di codice che possono essere inseriti in un sorgente in maniera automatica quando si digita la chiave corrispondente seguita dal tasto <TAB>. Molti prevedono dei segnaposto in cui si sposta il cursore per permettere la digitazione personalizzata. Il solito <TAB> conferma la personalizzazione e fa passare il cursore al prossimo segnaposto se c'è.

L'ordine delle chiavi nella tabella seguente rispecchia l'ordine con cui sono trattate le istruzioni negli appunti sul C++.

Chiave	*Codice generato*
io	`#include <iostream>`
std	`using namespace std;`

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

cout	`cout << "string " << var << endl;`
cin	`cin  >> var;`

if	Permette di scegliere fra:
	`if (condition){`
	`};`
	e
	`if (condition){`
	`}`
	`else {`
	`};`

?	`(cond) ? then : else;`
while	`while (condition){` `};`
for	`for (i=0; i<N; i++) {` `};`
do	`do {` `} while (condition);`
cast	`var = static_cast<Type>(var) ;`
inc	Permette di scegliere fra: `#include "..."` e `#include <...>`
vec	`vector<Class> var;`
iter	`vector<int>::iterator it;`
fori	`for (it=var.begin(); it!=var.end(); it++){` `};`

Chiave	*Codice generato*

f	Permette di scegliere fra:
	function:
	`type name(args)`
	`{`
	`};`
	fun_declaration:
	`type name(args);`
	f_method:
	`type Class::name(args)const`
	`{`
	`};`

struct	`struct name [` `};`

class	`class Name {`
	`public:`
	`protected:`
	`};`

<<	Permette di scegliere fra:
	operator<<:
	`ostream& operator<<(ostream& s,const type& c)`
	`{`
	`  return s;`
	`}`
	d_operator<<:
	`friend ostream& operator<<(ostream&,const Class&);`

>>	Come il precedente ma per l'operatore di estrazione >>

ifndef	`#ifndef NOMEBUFFER`
	`#define NOMEBUFFER`
	
	`#endif /* NOMEBUFFER */`

temp	Permette di scegliere fra:
	typename:
	`template <typename T>`
	class:
	`template <class T>`

open	`stream.open("file",ios::mode);`
close	`stream.close();`

Appendice 2: riferimenti

In questi appunti si fa riferimento alla versione 24 di Emacs configurata in coerenza alle indicazioni contenute in:

`http://ennebi.solira.org/cpplinux/configEmacs.zip`

la configurazione, già pronta e comprensiva di tutto, qui utilizzata.

`http://truongtx.me/2013/03/10/ecb-emacs-code-browser/`

la configurazione base da cui deriva quella cui si fa riferimento. Nel sito è specificato anche come fare per installare il pacchetto ECB.

`http://www.yolinux.com/TUTORIALS/src/dbinit_stl_views-1.03.txt`

la configurazione da usare nel file `.gdbinit` perché si possano usare nel debugger GDB i comandi per la visualizzazione dei containers delle STL

`http://www.emacswiki.org/emacs/SiteMap`

il sito di riferimento per conoscere il *mondo Emacs*, l'uso e le sue configurazioni.