Introduzione al C e C++

[*Quando ho iniziato a scrivere questo articolo, raccogliendo l'invito di Germano Rizzo, avevo molti dubbi; sopratutto sulla necessità, e l'opportunità, di scrivere l'ennesimo articolo sul C e C++. Una volta dissipati tali dubbi mi sono reso conto che, creare un qualcosa per persone a digiuno dell' argomento, era impresa più difficile di quanto avevo previsto. Soprattutto per la tendenza, propria di qualsiasi programmatore, a dare molte cose per scontate. Ho, quindi, deciso di chiedere l'aiuto di un caro amico: l'ing. Diego Parruccia: programmatore professionista e profondo conoscitore di C e C++. Ciò che leggerete ha superato la sua revisione e spero sia per Voi una lettura utile e/o gradevole...]*

Sommario

1) La storia del C

Il C è un linguaggio di programmazione sviluppato da Dennis Ritchie, ai laboratori Bell Labs, a metà anni 70. Venne chiamato C in quanto era un evoluzione del linguaggio B che, a sua volta, derivava dal BCPL. Data la sua diffusione, il linguaggio, è stato sottoposto ad un processo di standardizzazione portando all'approvazione dello standard, da parte dell' ANSI, nel 1989 (ANSI: X3.159-1989 / ISO: JTC1 SC22 WG14 e successive revisioni).
Il C, originariamente, era stato progettato come linguaggio di sistema e tale natura è ancora presente in molte delle sue capacità. Esso C viene definito un linguaggio di programmazione ad alto livello ma è, per sua origine, più vicino al codice assembly, e quindi al linguaggio macchina, di qualsiasi altro linguaggio ad alto livello. Ad oggi, probabilmente, il più grosso programma scritto in C è stato il sistema operativo UNIX e per molti anni il C è stato così strettamente legato a UNIX che le persone accomunavano i due termini.

2) C: Informazioni di base

Il nucleo del C è costituito da un insieme relativamente limitato di istruzioni: non comprende infatti funzioni per l'accesso alla memoria, per l'input/output o per il trattamento delle stringhe.
Tali funzioni, aggiuntive, sono fornite dalle librerie ed in particolare dalle librerie standard del C. Per poter usufruire di tali librerie (e quindi di tali funzioni) è necessario *"includerle"* nel programma principale. Un vantaggio di tale organizzazione è la possibilità di includere solo le librerie che servono ottenendo programmi molto più snelli e leggibili. E`, inoltre, possibile sfruttare questo modo di lavoro per creare le proprie librerie personalizzate e sostituire o integrare quelle standard.
Le funzioni vengono "dichiarate" o "definite" attraverso dei files detti headers (letteralmente *intestazioni*), che hanno estensione **.h** (anche se il *"corpo"* delle funzioni vere e proprie si trova nei files **.c** o precompilato nei files **.o** o **.lib**).
Gli headers vengono poi inclusi nel programma principale tramite un' apposita istruzione (#include) che esamineremo in seguito.

Un programma non banale è quindi formato da più files, con estensioni .c e .h, connessi tra loro da apposite chiamate, e la sua struttura generica è la seguente:

1. Direttive per il pre-processore (compresa inclusione di librerie e di altri files)
2. Definizione delle costanti
3. Definizione dei tipi
4. Funzioni
5. Funzione main (ovvero corpo principale del programma)

A questo punto, prima di addentrarci ulteriormente nei meandri del C, è opportuno almeno un esempio di base.

3) Esempi di C

Premetto che l'esempio che segue non brilla per originalità, ma serve in ogni caso a rendere un po' meno astratte alcune spiegazioni.
Con un editor di testo scrivete quanto segue in un file e salvatelo con il nome `miofile.c`

```
/*    Questo è un commento       */
/*-------------------------------*/
/* Nome del programma: miofile.c */

/*includo la libreria standard per l' I/O  */
#include <stdio.h>

/*creo la funzione principale del programma */
 void main ()
{
/* Emetto un messaggio sullo schermo. */
   printf ("Ciao mondo!\n");
}
```

Vi faccio notare i commenti, che vengono racchiusi fra due coppie di simboli (*/** e ***/**), e l'istruzione **`#include`** che carica l'header della libreria stdio.h (ovvero la libreria standard con le funzioni di I/O). Tale libreria è stata inclusa in quanto per stampare il messaggio abbiamo utilizzato la funzione *printf()* che è, appunto, compresa all'interno della libreria stdio.h. Tale istruzione non è un comando C ma una direttiva al preprocessore che indica di caricare il file aggiuntivo. In generale, all'interno dell' header, vi saranno le dichiarazioni dele funzioni della libreria e/o l'istruzione di caricamento di ulteriori files componenti la libreria.

Dopo quest' ulteriore digressione passiamo alla compilazione del programma. Dando per scontato che il compilatore sia gcc scrivete la riga seguente al prompt dei comandi (la seconda riga mostra l'output del compilatore):

```
$ gcc -Wall -o mioC mioC.c
mioC.c:11: warning: return type of `main' is not `int'
```

notate l'opzione -o con la quale assegno un nome, di mia scelta, al file eseguibile che viene compilato. Il nome di default è a.out e personalmente preferisco specificare sempre il nome. Avendo attivato la funzione -Wall, inoltre, il compilatore ci segnala tutti i warning ovvero gli avvertimenti (sugli errori), pertanto la riga di output ci segnala che la funzione main non ritorna un valore intero ma vuoto. Essendo questa una mia esplicita scelta (per dimostrare la funzionalità descritta) ignoriamo l'avviso e proviamo ad eseguire il programma.

```
$ ./mioC
Ciao mondo!
```

Orribile vero?
Andiamo comunque ad esaminare come funziona, a grandi linee, la compilazione; anche perchè finora ho dato per scontato che tutti sapessero che il C è un linguaggio compilato e che tutti fossero a conoscenza della differenza tra un linguaggio compilato ed uno interpretato. Innanzitutto un programma scritto in un linguaggio interpretato, al momento dell'esecuzione, viene letto da un interprete, ovvero un programma in codice macchina (o come si usa dire binario), che è in grado di comprendere le istruzioni del sorgente ed eseguirle. Un linguaggio compilato viene, invece, trasformato dalla sua forma nativa ad una forma direttamente eseguibile. Risulta ovvio che, un linguaggio compilato, permette di ottenere programmi più veloci di un equivalente codice interpretato in quanto, durante l'esecuzione, viene saltato il processo di interpretazione . Le fasi della compilazione sono le seguenti:

1. innanzitutto il compilatore interpreta il file sorgente .c nel preprocessore ed espande tutte le macro e le direttive comprese le #include
2. il codice sorgente risultante dalla fase di preprocessing viene compilato in un file di codice oggetto .o
3. viene quindi richiamato il linker che eseguendo il collegamento di tutti i files .o e .lib crea il file eseguibile

Su suggerimento del mio supervisore preferito vorrei spendere due righe sui vari tipi di files che vi trovate fra i piedi lavorando con il C:

- files **.c** contengono il codice sorgente in c
- files **.h** come già accennato contengono le intestazioni delle funzioni se necessario possono richiamare altri file .h dal loro interno
- files **.i** cntengono il codice sorgente pre-processato
- files **.o** e **.lib** contengono il codice oggetto che viene passato al linker

A questo punto evito di ricreare l'ennesimo corso sul C e vi rimando alla sezione, dedicata a tale argomento, degli Appunti di informatica libera di Daniele Giacomini [No, non siamo parenti :-)]. Una lettura che ho trovato fondamentale in molte occasioni.

Riporto solamente una nota curiosa:
Sembra che, in assoluto, l'errore più frequente, commesso programmando in C, sia quello di dimenticare un segno di uguale (=) all'interno delle operazioni di confronto. Vediamo un esempio: l'istruzione

```
if (i==3)
```

esegue il confronto fra la variabile i ed il numero 3. Scrivendo invece

```
if (i=3)
```

si assegna ad i il valore 3 ed il test risulta sempre vero. Per evitare quest' errore, banale ma frequentissimo, i guru della programmazione suggeriscono di utilizzare nei test la forma seguente:

```
if (3==i)
```

anteponendo il valore costante alla variabile. Infatti se in tale forma si omette un segno = il compilatore genera una segnalazione di errore in quanto stiamo tentando di assegnare ad un numero costante una variabile.

Per ulteriori approfondimenti sul C vi rimando nuovamente ai riferimenti bibliografici [servono a questo :-) vero?] e sposto l'attenzione sul C++

4) La Storia del C++

Il C++ nasce all'inizio degli anni ottanta, all'interno dei laboratori Bell, grazie all'opera del matematico Bjarne Stroustrup che era stato incaricato di sviluppare un software per simulare il funzionamento di alcuni centralini di commutazione telefonica. L'applicazione richiedeva un linguaggio con alte prestazioni ed il C, era considerato il più adatto ma, dall'analisi, risultava la necessità di un approccio ad oggetti e fu, quindi, creato un superset del linguaggio C chiamato inizialmente "C with classes" che soddisfaceva le necessità del caso.

Dal 1986 il linguaggio iniziò a diffondersi e ad essere utilizzato, in maniera sempre maggiore, al di fuori del gruppo di programmatori cui faceva capo Stroustrup. Il linguaggio C++ fu, poi, sottoposto al processo di standardizzazione del comitato ANSI (American National Standard Institute), a partire dal 1989, con il numero di protocollo X3J16 e divenne standard da settembre 1994.

Uno degli scopi principali del C++ era quello di mantenere piena compatibilità con il C. L'idea era quella di conservare l'integrità di molte librerie C e l'uso degli strumenti sviluppati per il C.

5) Il Linguaggio C++

Il miglioramento più rilevante introdotto dal linguaggio C++ è il supporto della programmazione orientata agli oggetti (Object Oriented Programming: OOP). Tale metodo permette di ottenere un codice facile da mantenere e riutilizzare. Per sfruttare tutti i benefici introdotti dal C++ occorre quindi passare da una programmazione strutturata ad un approccio ad oggetti.

Tentiamo, quindi, di definire che cosa si intende per oggetto. Innanzitutto non si deve cadere nell'equivoco che si possa definire oggetto solo un qualcosa di solido. In programmazione tale termine può definire tanto un qualcosa di tangibile quanto un concetto. Per rendere, però, le cose più chiare potrei definire un oggetto come una variabile nella quale vengono immagazzinati i dati. Rispetto ad altre variabili l'oggetto ha la caratteristica di essere dotato di proprietà e di interfacce.
Un esempio di oggetto (come presentato da Bruce Eckel:) può essere la lampadina. Essa ha delle proprieta come il suo stato (in un dato momento può essere accesa o spenta), il colore, la tensione di funzionamento, ecc. Possiede anche delle interfacce in quanto può venire accesa o spenta.
In un programma C++ l'oggetto è rappresentato dalla *classe*, le proprietà dalle *variabili private* della classe e le interfacce dalle *funzioni pubbliche*. Tutto ciò verrà ulteriormente chiarito da quanto verrà esposto di seguito.

La programmazione orientata agli oggetti (OOP) si basa su tre concetti fondamentali: incapsulamento, ereditarietà e polimorfismo.
L'incapsulamento è la capacita di raccogliere dati e funzioni nelle classi per nascondere il funzionamento e l'eleborazione, interna alla classe stessa, ed esportare solo un' interfaccia che permetta l'interfacciamento con il resto del programma.
L'ereditarietà è la capacità di acquisire le strutture dati e funzioni di altre classi di oggetti (classi base) in modo da riutilizzarli in nuovi classi di oggetti (classi derivate).
Per ultima la capacità, forse, più importante, il polimorfismo, che è la capacità di alcune funzioni (chiamate virtuali) di svolgere a run-time funzioni diverse a seconda della classe cui fanno riferimento. Per non appesantirlo troppo, in questo articolo non approfondiremo questi concetti. Potete comunque fare riferimento alla bibliografia per ulteriori informazioni.

Cercherò di dare alcuni esempi di codice che illustrano i concetti finora riportati:

6) Esempi di C++

Tentiamo di realizzare qualcosa di più rispetto al solito "Ciao mondo" (anche se il tema rimarrà lo stesso).
Il programma che segue è scritto in C++ ma in forma strutturata anziché ad oggetti. Produce due scritte sullo schermo (in modalità testo). Chiaramente la semplicità di quest'esempio non richiederebbe alcuna implementazione ad oggetti, ma noi andremo poi a confrontare le due forme di scrittura.
Con un editor di testo scrivete quanto segue in un file chiamato `primo.C`

```
//primo.C -questo è un commento con il nome del programma
//includiamo la libreria per l'IO
#include <iostream>
//iniziamo con il programma principale (si chiama sempre main)
void main (int argc,char *argv)
 {
  char *soggetto;
  char *saluto;
//un saluto dall'autore
  soggetto="L'autore dice";
  saluto=" - Ciao mondo - ";
  cout << soggetto << saluto << endl;

//un saluto dal pc
  soggetto="Il pc dice";
  cout << soggetto << saluto << endl;
 }
```

Come prima cosa faccio notare che l'estensione del file è .C (e non .c) in quanto tale estensione è quella attribuita di default ai files C++ (a seconda del compilatore altre estensioni valide sono .cc e .cpp). poi vorrei mettere in evidenza che ora i comenti sono delimitati da una doppia barra // all'inizio della riga (anche se la forma di commento del C è ancora utile per commentare blocchi di codice). Inoltre l'output del programma non è più affidato alla funzione printf() ma all'operatore di stream (flusso) << che è incluso nella libreria iostream.h. A questo punto compiliamo il programma. Do per scontato che stiate usando Linux con installato il compilatore g++ e che vi siate già portati nella directory contenente il file sorgente.

```
$ ./g++ -g -o primo primo.C
```

eseguite il programma digitando il nome:

```
$ ./primo
```

il risultato sarà qualcosa di simile
```
L'autore dice - Ciao mondo -
Il pc dice - Ciao mondo -
```

Ora trasformiamo il tutto in un programma ad oggetti cercando di introdurre alcune nozioni

Ci sono alcune regole che sarebbe bene seguire:

- In generale le classi dovrebbero essere definite nei files di intestazione (.h).
- Le funzioni membro che sono definite all'interno di una definizione di classe possono essere inserite nei files di intestazione.
- Le funzioni esterne ad una classe di solito sono inserite nei files .C o .cpp (a seconda delle convenzioni del compilatore).

Generalizzando di solito per una classe C++ si ha la seguente struttura:

```
Classe
  Area privata:
       variabili
       funzioni di servizio

  Area pubblica:
       metodi (funzioni pubbliche)
       overloading di operatori
       variabili pubbliche (meglio evitarle sono una cattiva abitudine)
```

vediamo quindi la re-implementazione del programma precedente:

```
//secondo.C -questo è un commento con il nome del programma
//includiamo la libreria per l'IO
#include <iostream>
#include <string>
//dichiariamo la classe saluto
 class saluto {
   private:
     char *soggetto;
     char *messaggio;
//una funzione per inizializzare le stringhe
     void init_saluto(const char  *sogg,const char *mess){
        soggetto = new char [strlen(sogg)+1];
        strcpy(soggetto,sogg);
        messaggio = new char [strlen(mess)+1];
        strcpy(messaggio,mess);
     }

   public:
     //il costruttore della classe
     saluto(const char *sogg, const char *mess){
         init_saluto (sogg,mess);
     }

     //il distruttore della classe
     ~saluto(){
               delete (soggetto);
               delete (messaggio);
               }

     //la funzione pubblica di visualizzazione
     void  visualizza () const {
        cout << soggetto << messaggio << endl;
     }
  };

//iniziamo con il programma principale (si chiama sempre main)

  void main (int argc,char *argv) {

    saluto sal1 ("L'autore dice"," - Ciao mondo - ");
    saluto sal2 ("Il PC dice" ," - Ciao mondo - ");

    sal1.visualizza();
    sal2.visualizza();
    }
```

dopo la compilazione e l'esecuzione, naturalmente, il risultato è identico a quello dell'esempio precedente. Il primo programma contiene 10 righe di codice, esclusi i commenti, il secondo 29 ovvero quasi tre volte tante.
Fra i vari motivi, di aumento di dimensione, vi sono le funzioni del costruttore e distruttore della classe: Il costruttore è una funzione con lo stesso nome della classe ed è la prima funzione membro chiamata da un oggetto quando viene creato in memoria. Può essere usata per inizializzare i dati membro dell' oggetto stesso.
Il distruttore è l'ultima funzione membro chiamata da un oggetto. Questa funzione viene chiamata prima della rimozione del oggetto dalla memoria. Generalmente viene utilizzata per liberare le risorse occupate dall' oggetto (memoria, files, risorse di sistema, ecc.). Sostanzialmente è una funzione con lo stesso nome della classe preceduto da una tilde [~]. Nel esempio precedente possiamo vedere l'uso del costruttore *saluto(const char *sogg, const char *mess)* e del distruttore *~saluto()*.
Vi è un ovvia considerazione che nei programmini così piccoli l'overhead creato dallo scrivere codice a oggetti sembrerebbe rendere pesante, se non addirittura sconsigliabile, questo tipo di approccio.

Confutiamo subito tale affermazione partendo da un esempio pratico anche se banale: Il vostro cliente vi ha appena telefonato e annuncia che vuole invertire l'ordine delle scritte in uscita ovvero il risultato dovrà essere:

```
- Ciao mondo - L'autore dice
- Ciao mondo - Il PC dice
```

nel primo programma dovrete modificate le due righe che producono l'output:

```
cout << soggetto << saluto << endl;
```

diventa

```
cout << saluto << soggetto << endl;
```

nel caso della programmazione ad oggetti tale modifica viene operata una volta sola all'interno della funzione visualizza().

In realtà anche il primo esempio poteva essere scritto, in maniera migliore, in modo da richiedere un unica modifica. Ho voluto banalizzare l'esempio per far comprendere come un codice ad oggetti può rivelarsi molto più facile da mantenere.

7) C, C++: quando e perché...

La scelta di un linguaggio di programmazione è talmente personale e legata a tanti fattori che consigliarne uno specifico è una cosa veramente ostica. Oltretutto la, seppur breve, storia dell'informatica c'insegna che non sempre il linguaggio migliore, dal punto di vista tecnico è, il più utilizzato. Si veda in tal proposito il confronto fra linguaggi di David Wheeler (www.adahome.com/History/Steelman/steeltab.htm) nel quale l'ADA ottiene una valutazione del 93%, Java 72%, C++ 68% ed il C ottiene il 53%. Inoltre, secondo alcune analisi, il costo di sviluppo, di un programma, in ADA e pari alla metà del costo del programma equivalente in C++. Un'altra cosa da notare è che un compilatore C++ è notevolmente più complesso di un compilatore C ed i programmi in C++ sono più lenti dei programmi equivalenti in C (anche se queste differenze con i processori attuali e con il costo attuale della RAM non sono certamente così evidenti). Questo è dovuto al fatto che i compilatori C++ sono ancora "immaturi" rispetto ai compilatori C.

Di fronte a queste osservazioni sembra che il C++ abbia solo connotazioni negative. In realtà queste premesse sono necessarie per avere un quadro globale della situazione. Sono dell'opinione, infatti, che non esiste un linguaggio di programmazione migliore di altri ed ognuno ha vantaggi e svantaggi che lo rendono più o meno utile a seconda del tipo di applicativo che si deve realizzare. Oggigiorno, ad esempio, il C viene usato soprattutto per la programmazione di basso livello come i Sistemi Operativi [si anche Linux :-)] e i device drivers.

Il C++ consente lo sviluppo di software su larga scala. Grazie a un maggiore rigore sul controllo dei tipi, molti degli effetti collaterali tipici del C e molti errori, divengono impossibili in C++ e grazie agli oggetti il codice C++ è (se scritto bene) sufficientemente portatile e facile da mantenere da permettere lo sviluppo di grosse applicazioni.

Una delle principali guerre di "religione dei linguaggi" dei nostri giorni mette a confronto il C++ con Java. Pur non volendo prendere una posizione a favore dell'uno o dell'altro cercherò di riportare alcune notizie fra quelle più diffuse:

Il C++ genera programmi estremamente veloci ed è, da 10 a 20 volte, più veloce di JAVA non solo quando Java viene interpretato, ma anche quando Java viene compilato da un JIT (just in time compiler).
Java viene usato molto in quanto la gestione della memoria è automatizzata ed il programmatore non deve preoccuparsi dell'allocazione della RAM;

Il C++ è un sovrainsieme del C e quindi mantiene anche le caratteristiche peggiori del C. Prima fra tutte l'allocazione di memoria manuale che oltre a essere noiosa da gestire è una delle principali fonti di errori. In questo senso Java è imbattibile a meno di non avere sottomano delle ottime librerie di gestione della memoria.

Se ciò di cui si necessita è un linguaggio di scripting come Javascript allora si può puntare sul PHP che ha una sintassi derivata da quella del C++. Javascript è indipendente dalla piattaforma ma molto lento come linguaggio di scripting. Se avete accesso ad un server con PHP abilitato esso offre maggiori prestazioni. Inoltre poiche esso lavora dal lato server avrete il controllo della situazione in maniera maggiore rispetto a quanto permette Javascript. Questo perchè J.S. lavora dal lato client e non è mai possibile sapere a priori quale web browser viene utilizzato mentre lo script sul lato server lavorerà sempre come Voi avete deciso.

8) Gli strumenti

Quello che segue è poco più di un elenco di una serie di strumenti, naturalmente Open Source, che utilizzo per il mio lavoro (o comunque ho trovato degni di nota). Ognuno di Voi mi perdoni, se non ho citato il suo editor o compilatore preferito, ma ho preferito descrivere solo strumenti che ho provato almeno una volta...

Compilatori:

Gcc GNU Compiler Collection; ovvero lo strumento GNU per eccellenza: il Compilatore con il quale vengono compilati tutti i progetti Linux. E` in grado di compilare C, C++,Objective C, Java, Fortran e ADA su oltre 40 diverse famiglie di chips, ed è corredato delle librerie per questi linguaggi.

GNU BloodShed per compilare con Windows, si basa sostanzialmente sul gcc.

MSDOS C++ compiler per chi ha la necessità di compilate in ambiente MS DOS (Nota un link check mi ha indicato che il link non è più valido - accetto segnalazioni in merito).

Librerie:

Come avevo già sottolineato esistono numerose librerie che risolvono i limiti di quelle standard del C e del C++. Alcune di queste sono corredate di ambienti di sviluppo più o meno integrati. Accenno di seguito alle più usate:

Iniziando dalla mia preferita: FLTK (si pronuncia "fulltick") è un toolkit (ovvero un insieme composto da ambiente grafico di sviluppo, libreria ed editor) per il C++ multi piattaforma. Ad oggi lavora con X (praticamente tutti gli UNIX), MacOS, e MS Windows. Supporta la grafica 3D tramite le librerie OpenGL.
Lo considero veramente portatile come ambiente di sviluppo anche se va integrato con un editor aggiuntivo in quanto quello in

dotazione è ancora scarno. Il suo vantaggio principale è quello di portare a generare eseguibili di dimensioni ridotte e alta velocità.

Un'altra libreria da me molto usata in passato e che ora ho abbandonato è la XForms un toolkit basato sulla xlib (e quindi su X-window) con un editor di interfaccia in grado di generare il relativo codice C. Soffre del problema della scarsa portabilità ma il codice generato è molto snello.

La libreria GTK+ è un toolkit multi piattaforma orientato allo sviluppo in C anche se esistono le implementazioni delle interfacce per il suo utilizzo tramite C++.
Inizialmente era solo la libreria tramite la quale è stato sviluppato il noto software di grafica Gimp poi è stata adottata per lo sviluppo del Window Manager Gnome e si è estesa alle altre applicazioni Gnome. Fra le varie applicazioni vi è anche un editor per l'interfaccia grafica Glade che semplifica notevolmente il lavoro di costruzione della stessa producendo il codice C relativo.

La libreria Qt è una libreria, orientata al C++, multi piattaforma, che supporta MS/Windows, Unix/X11 (Linux, Sun Solaris, HP-UX, Digital Unix, IBM AIX, SGI IRIX e altre varianti di Unix),Macintosh (compreso Mac OS X) e piattaforme Embedded. tale libreria è diffusa in quanto è alla base dello sviluppo del window manager KDE che oggi assieme a Gnome è uno dei Wm preferiti degli utenti Linux.
La libreria è accompagnata da un IDE (Integrated Development Enviroment) per UNIX, KDevelop, che possiede delle buone caratteristiche e che, come il resto dell'ambiente, ha rilasci frequentissimi con notevoli migliorie.
La Qt non gode, comunque, di buona fama presso la Open Community in quanto inizialmente era stata rilasciata con una licenza più restrittiva della GPL e quindi era guardata con sospetto.

Editors

Xwpe è un editor per UNIX con una interfaccia compatibile con quella dei vecchi editor del Borland C e Borland Pascal per DOS. Da xwpe potete editare il programma, avviare la compilazione e il debug dei programmi. L'estetica è sicuramente un po' datata, anche perchè lo sviluppo del programma aveva subito una battuta d'arresto, ma le varie funzionalità di cui è dotato lo rendono insuperabile (a mio parere chiaramente).

Chiamare editor Emacs è estremamente riduttivo. Questo programma scritto da R. Stallman è un ambiente di lavoro in cui si può fare di tutto da leggere la posta a compilare un programma. ella sua home page viene definito come *editor realtime estensibile, personalizzabile e autodocumentante*. Troverete centinaia (se non migliaia) di utilizzatori di Emacs pronti a giurare che è il miglior editor per qualsiasi linguaggio di programmazione.

Se, invece, avete la necessità di usare lo stesso editor qualsiasi sia la piattaforma su cui sviluppate, sappiate che VI o meglio la sua versione migliorata VIM gira su AmigaOS, AtariMiNT, BeOS, DOS, MacOS,MachTen, OS/2, RiscOS, VMS, e Windows (95/98/NT4/NT5/2000/XP/.NET/64-Bit/Embedded)e, naturalmente, tutte le versioni di UNIX come affermato nella home page del programma.
Personalmente lo utilizzo spesso per amministrare i sistemi UNIX di cui sono responsabile ma non lo trovo funzionale come xwpe come ambiente di programmazione.

Vorrei chiudere con una considerazione. Tutti i programmatori, me compreso, hanno la tendenza a riscrivere codice da zero. Questo serve, nella fase iniziale, per crearsi un insieme di esperienze dirette (si impara molto dai propri errori). Ma come dice, giustamente, Eric Raymond nel suo "La cattedrale e il bazar" ogni programmatore ha il dovere di evitare di reinventare la ruota e, quindi, di riutilizzare il codice già scritto da altri sviluppatori e testato dalla comunità. Questo immenso patrimonio di codice già scritto che l'Open Source porta con se dovrebbe quindi diventare la Vostra risorsa principale e la principale fonte di istruzione. Nel caso del C e ancor più nel caso del C++ il riutilizzo delle librerie e dei sorgenti è facilitato dalla natura stessa del linguaggio. Fatene buon uso e che la forza...

Riferimenti bibliografici:

C

C++ for Unix quick-reference, with C variations del Dr. David Wessels
Standard C (University of California, San Diego)
C and its immediate ancestors di Dennis M. Ritchie
Appunti di informatica libera di Daniele Giacomini

C++

Programmare in C++ di Stephen Blaha (Apogeo Ed.)
Thinking in C++ di Bruce Eckel
C++ Coding Standard di Todd Hoff
C and C++ in 5 days di Philip Machanick
C++ Programming How-To di Alavoor Vasudevan

di Rudi Giacomini Pilon
e Diego Fernando Parruccia