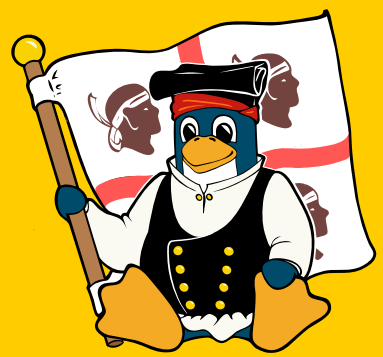
Portable and cross-development

Luca Cireddu, Roberto Metere

26 ottobre 2013



GULCh

Gruppo Utenti Linux Cagliari h...?

Sviluppo multipiattaforma

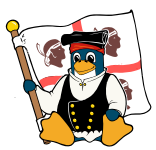
- Portabilità
- Problemi correlati

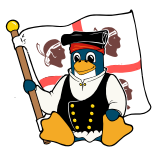
Scenari a confronto, problemi

- Analisi pre-stesura del codice
- Cross-development sul codice
- Librerie di terze parti
- Virtual Machine

Cross-compiling

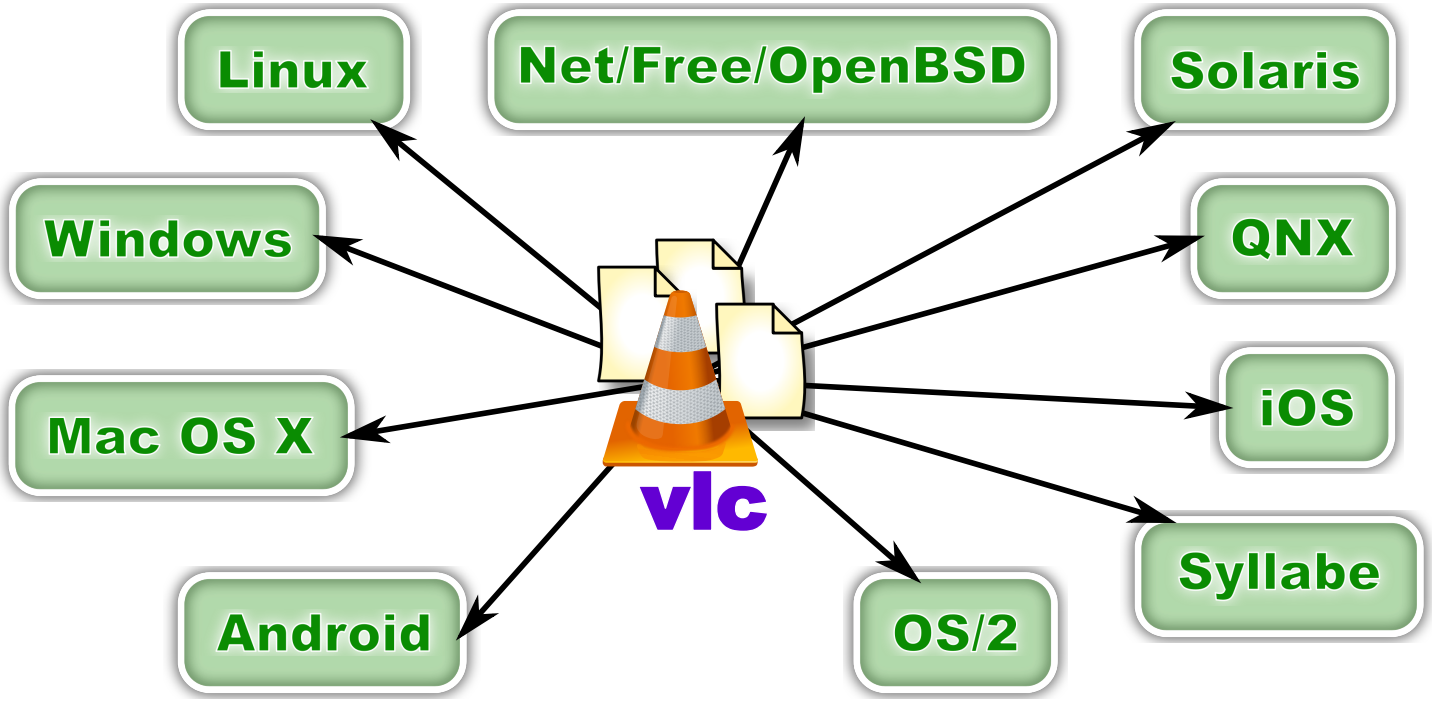
- Dall'host al target
- Creare l'eseguibile per il *target*





Portabilità

varie piattaforme

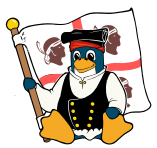


Portabilità

definizione

La portabilità è la caratteristica di un programma utilizzabile in sistemi (hardware e software) diversi senza subire modifiche.
È misurabile in due modi:

- percentuale di istruzioni dipendenti dal *target*
- numero di sistemi *target*



Portabilità

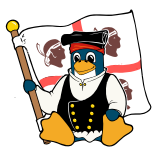
piattaforma a target

Una piattaforma si distingue da un'altra o perché ha un diverso hardware o perché ha un diverso sistema operativo.

La **piattaforma host** è quella nella quale risiede il lavoro di sviluppo del software, mentre la **piattaforma target** è quella nella quale il software finale verrà testato ed eseguito.

Le piattaforme spesso si indicano come coppie di SO/CPU, esempi:

- Linux/x86_64
- Solaris/UltraSPARC
- Windows/IA32
- MacOSX/PowerPC



Problemi correlati

principali aree organizzative legate alla portabilità

Analisi pre-codice 1

Problemi correlati

principali aree organizzative legate alla portabilità

Analisi pre-codice 1

Sviluppo cross sul codice 2

Problemi correlati

principali aree organizzative legate alla portabilità

Analisi pre-codice 1

Sviluppo cross sul codice 2

Cross compiling 4

Problemi correlati

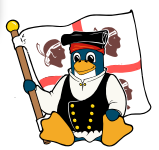
principali aree organizzative legate alla portabilità

1 Analisi pre-codice

2 Sviluppo cross sul codice

3 Librerie di terze parti

4 Cross compiling



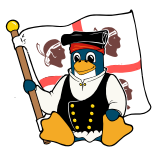
Problemi correlati

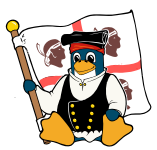
strategie di design

Politica di sviluppo

La risoluzione dei problemi di portabilità si articolano nelle seguenti strategie di design:

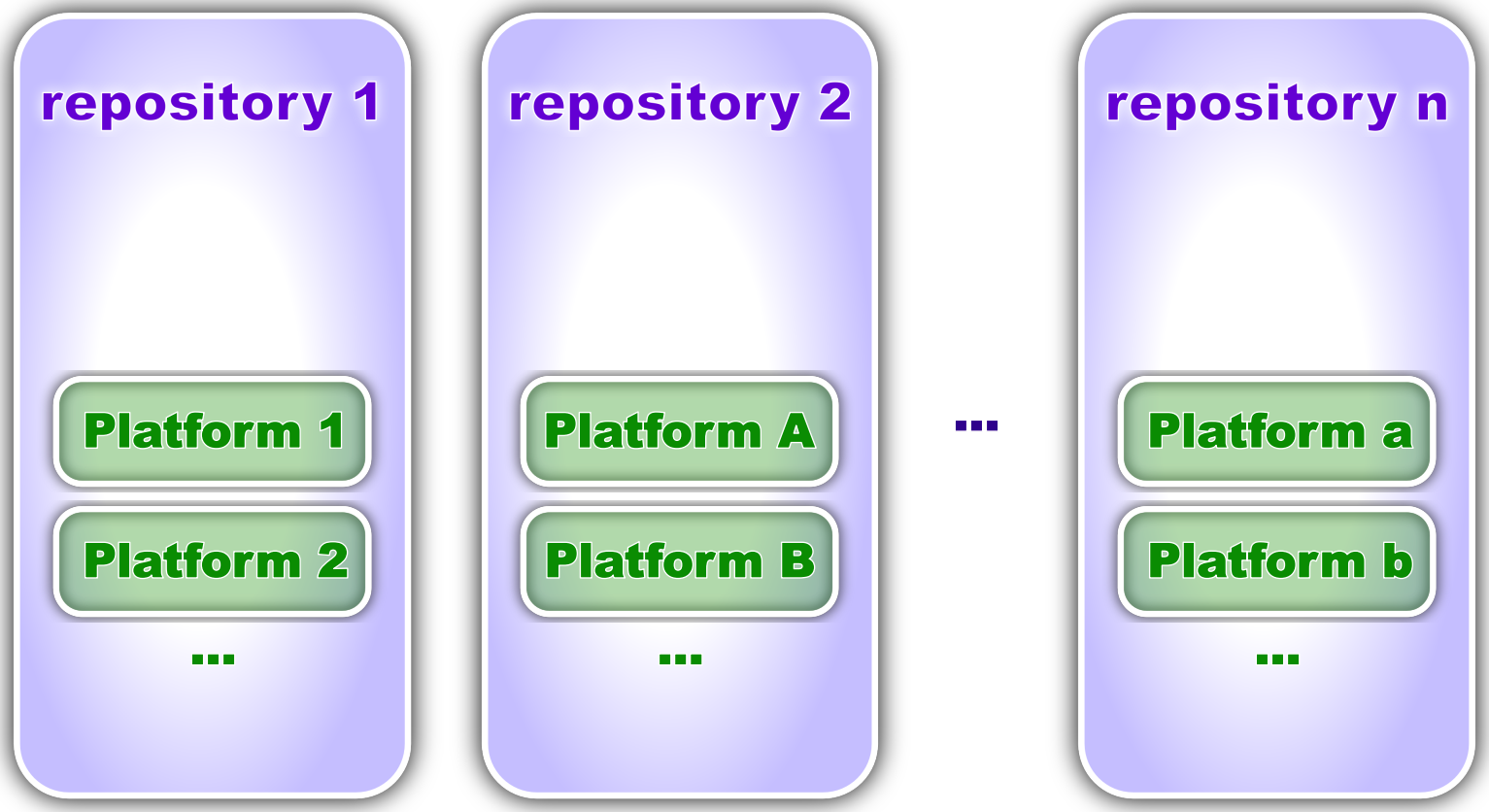
- *graceful degradation*
- separazione delle funzionalità
- più repository di sviluppo
- librerie di terze parti





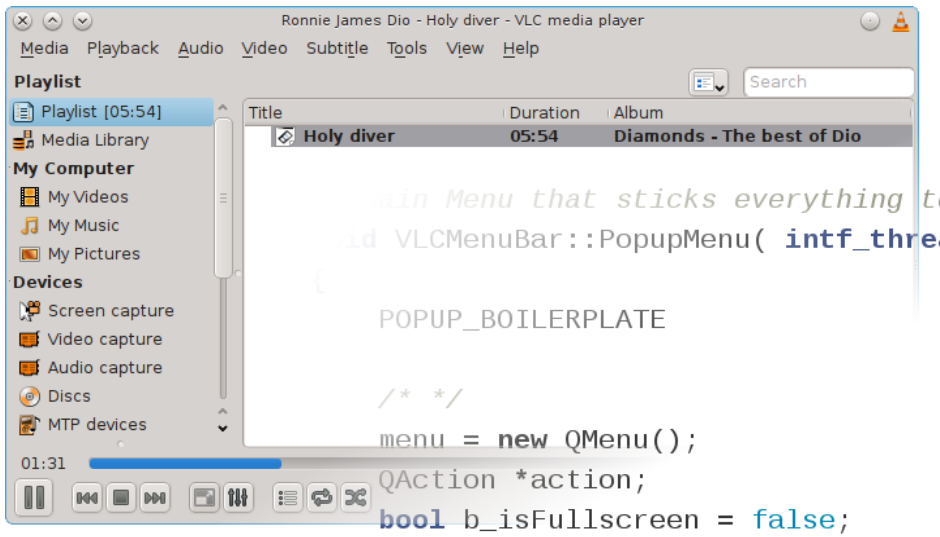
Analisi pre-stesura del codice

la mano al direttore del progetto



Analisi pre-stesura del codice

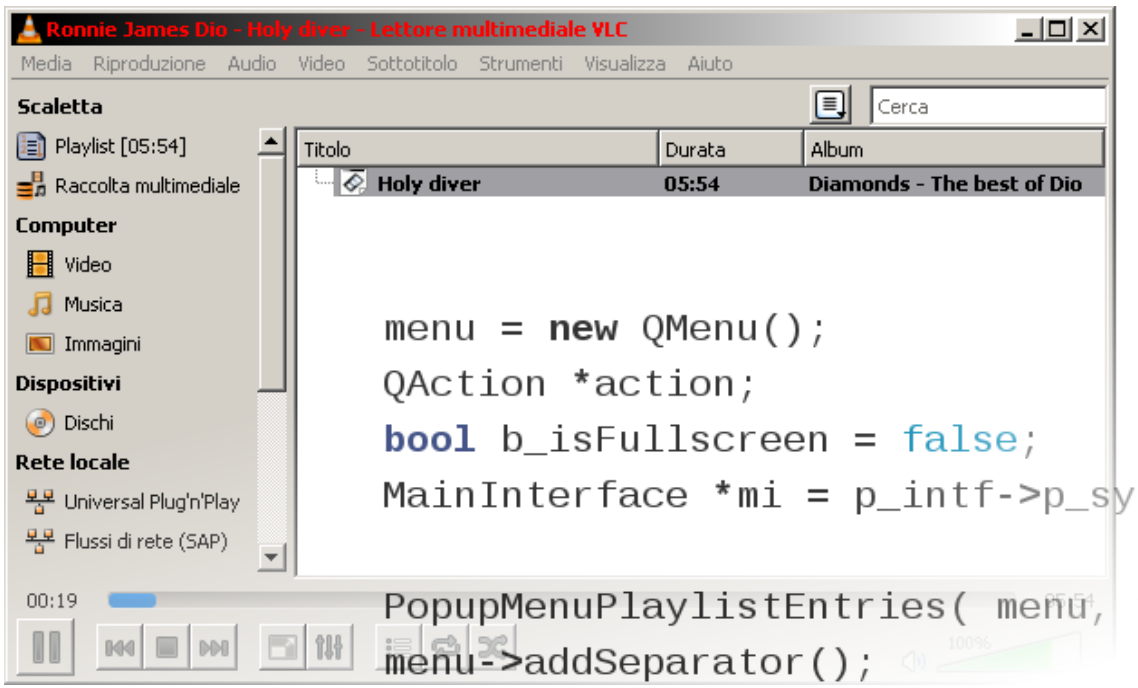
repository Linux/Windows + Qt




```
ain Menu that sticks everything together  */
id VLCMenuBar::PopupMenu( intf_thread_t *p_intf, bool show )
{

    POPUP_BOILERPLATE

    /* */
    menu = new QMenu();
    QAction *action;
    bool b_isFullscreen = false;
    MainInterface *mi = p_intf->p_sys->p_mi;

    PopupMenuPlaylistEntries( menu, p_intf, p_input );
    menu->addSeparator();
```

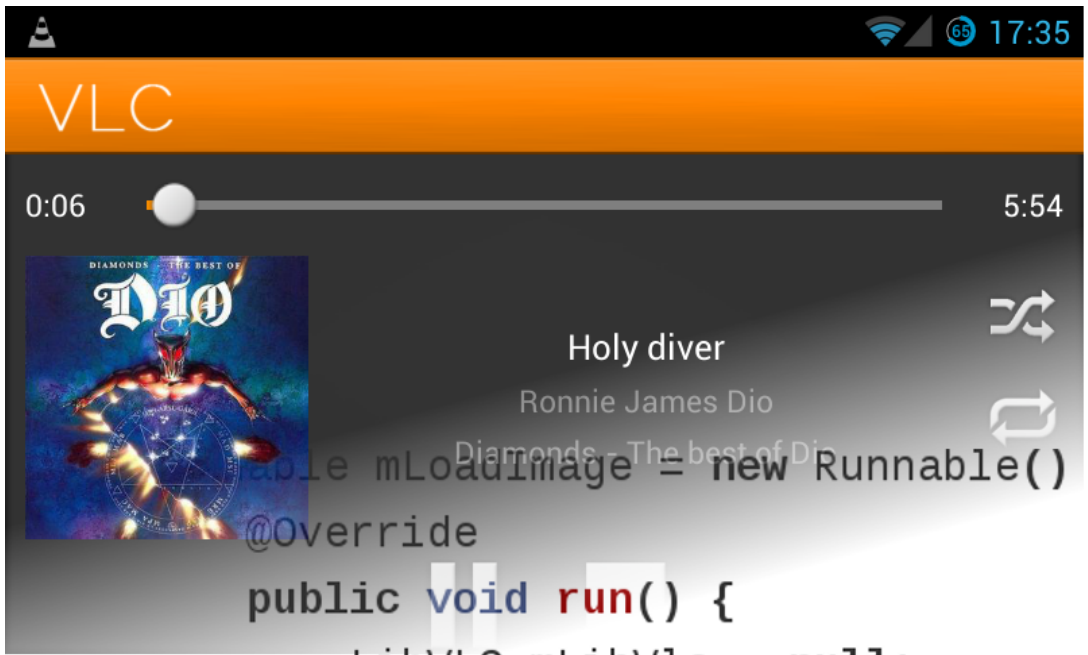
Analisi pre-stesura del codice

repository Linux/Windows + Qt



Analisi pre-stesura del codice

repository Android + Java




```
Runnable mLoadImage = new Runnable() {
    @Override
    public void run() {
        LibVLC mLibVlc = null;
        try {
            mLibVlc = Util.getLibVlcInstance();
```

Analisi pre-stesura del codice

pro e contro

I pro . . .

- sviluppatori con *skill* settoriali (Java, C#, . . .)
- test da fare in ogni *target*, ma modifiche mirate per *target*

. . . e i contro

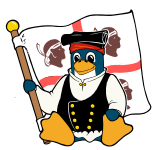
- risorse e costi (programmatori, tempo, . . .)
- difficile organizzare release contemporanee per tutti i *target*
- riusabilità del codice generalmente assente
- difficoltà per lo sviluppo di *plugin* cross-platform

Ma a volte può essere la scelta migliore, specialmente per i frontend.

Sviluppo multipiattaforma
- Portabilità
- Problemi correlati

Scenari a confronto, problemi
- Analisi pre-stesura del codice
- Cross-development sul codice
- Librerie di terze parti
- Virtual Machine

Cross-compiling
- Dall'host al target
- Creare l'eseguibile per il *target*



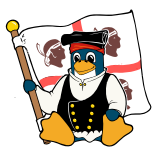
Cross-development sul codice

la mano al programmatore

La portabilità di un software per più sistemi *target*, in quest'area di sviluppo, è compito del programmatore.

Come?

- il codice viene adattato per i diversi *target*
- può essere **preprocessato** diversamente a seconda del *target*
- può contenere dei controlli a *runtime* sul *target*



Cross-development sul codice

esempio a compile time, in C

```c
#include <stdio.h>
int main() {
#ifdef WIN32
  printf("Siamo su Win32\n");
#else
  printf("Non siamo su Win32\n");
#endif
  return 0;
}
```

preprocessore C se compiliamo su Linux

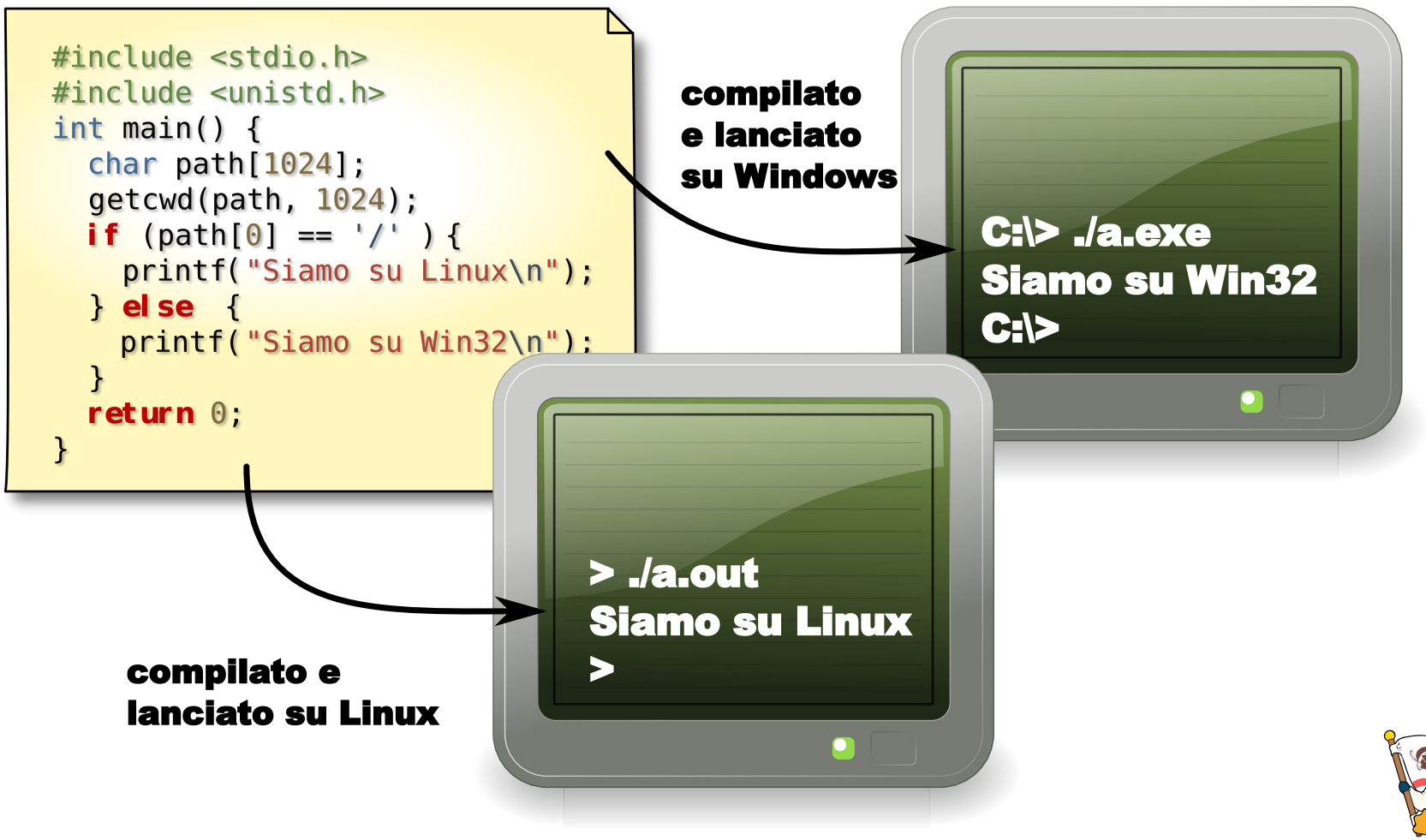
```c
#include <stdio.h>
int main() {
#ifdef WIN32
  printf("Siamo su Win32\n");
#else
  printf("Non siamo su Win32\n");
#endif
  return 0;
}
```

preprocessore C se compiliamo su Windows

```c
#include <stdio.h>
int main() {
#ifdef WIN32
  printf("Siamo su Win32\n");
#else
  printf("Non siamo su Win32\n");
#endif
  return 0;
}
```

Cross-development sul codice

esempio a runtime, in C



Cross-development sul codice

esempio a runtime

Il **javascript** delle applicazioni web è spesso un esempio di sviluppo cross sul codice.

Con dei controlli sul browser utilizzato, visto come un **interprete del javascript**, alcune parti vengono escluse a seconda di quanto un browser è aggiornato (per esempio) o, a seconda di quale browser si tratti, i codici che hanno un medesimo risultato possono essere diversi e incompatibili, quindi *protetti* da controlli (`if`, `switch`, ...).

Cross-development sul codice

pro e contro

I pro . . .

- buona parte del codice è comune, si ha un'alta riusabilità
- risorse e costi ridotti (meno programmatori)
- semplicità di rilascio delle release per tutte le piattaforme

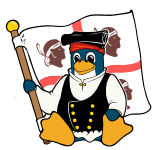
. . . e i contro

- i programmatori devono essere esperti nei vari *target* di competenza
- test in tutti i *target* "Write once, **debug** everywhere"
- codice generalmente più difficile da leggere
- le funzionalità non sono sempre le stesse (es. differenze del comportamento della `sleep`)

Sviluppo multipiattaforma
- Portabilità
- Problemi correlati

Scenari a confronto, problemi
- Analisi pre-stesura del codice
- Cross-development sul codice
- Librerie di terze parti
- Virtual Machine

Cross-compiling
- Dall'host al target
- Creare l'eseguibile per il *target*



Librerie di terze parti

una mano dagli altri

Se un codice cross-platform già funziona nelle piattaforme *target* di un nuovo progetto, perché non riutilizzarlo?

Riutilizzare il codice cross

- nasce naturalmente sotto forma di librerie
- le librerie (open) vengono distribuite (es. Qt, GTK, ...)
- **per le funzioni usate dalle librerie di terze parti, il programmatore non ha la necessità di badare al target**[1]
- esempi: Qt, GTK+, cairo, Mono, wxWidgets, Lazarus, haXe, Tcl/tk, ...

[1] almeno non troppo...

Librerie di terze parti

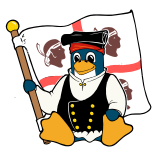
pro e contro

I pro . . .

- parte del cross-development è delegato alle librerie in modo trasparente
- semplicità di rilascio delle release per tutte le piattaforme
- codice più piccolo e facile da leggere, quindi anche test più semplici

. . . e i contro

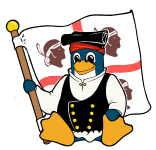
- il codice è fortemente orientato verso i pattern offerti dalle librerie e i loro vincoli
- è necessario che le librerie siano compilabili su tutti i *target* voluti
- a volte versioni diverse hanno funzionalità leggermente diverse per la stessa funzione (su codice), o la stessa funzione si comporta non proprio ugualmente nei vari *target* come ci si potrebbe aspettare



Sviluppo multipiattaforma
- Portabilità
- Problemi correlati

Scenari a confronto, problemi
- Analisi pre-stesura del codice
- Cross-development sul codice
- Librerie di terze parti
- Virtual Machine

Cross-compiling
- Dall'host al target
- Creare l'eseguibile per il *target*

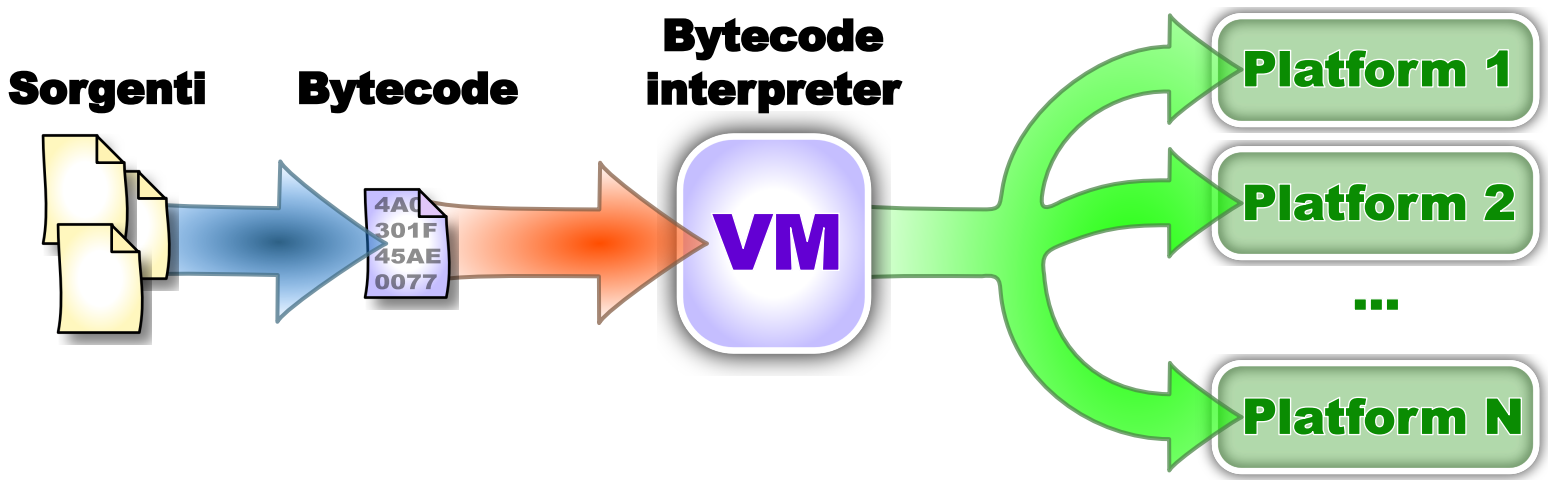


Virtual Machine

ne rimarrà soltanto uno

E se ci fosse un solo *target*?

- un unico target attraverso un processore virtuale (VM)
- chi programma la VM si preoccupa che funzioni in più sistemi *target* reali



Virtual Machine

pro e contro

I pro . . .

- il cross-development è delegato quasi al 100% alla VM
- "Write once, run everywhere"
- semplicità di rilascio delle release per tutte le piattaforme
- test facilitati
- codice facile da leggere

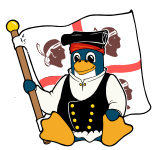
. . . e i contro

- rispetto ad altre soluzioni cross, l'applicazione può risultare più lenta
- è necessario che la VM sia compatibile con tutti i *target* voluti

Sviluppo multipiattaforma
- Portabilità
- Problemi correlati

Scenari a confronto, problemi
- Analisi pre-stesura del codice
- Cross-development sul codice
- Librerie di terze parti
- Virtual Machine

Cross-compiling
- Dall'host al target
- Creare l'eseguibile per il *target*



Dall'host al target

panoramica

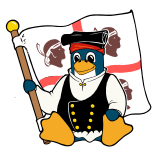
Semplicemente, compilare da una macchina per un *target* differente.

L'importanza del cross-compiling

- avviene quando il compilatore che viene lanciato in un sistema produce file binari per un'altro sistema
- necessario quando il sistema *target* non ha un insieme nativo di strumenti di compilazione
- utile quando in cui si compila è effettivamente più veloce del *target* o gode di maggiori risorse

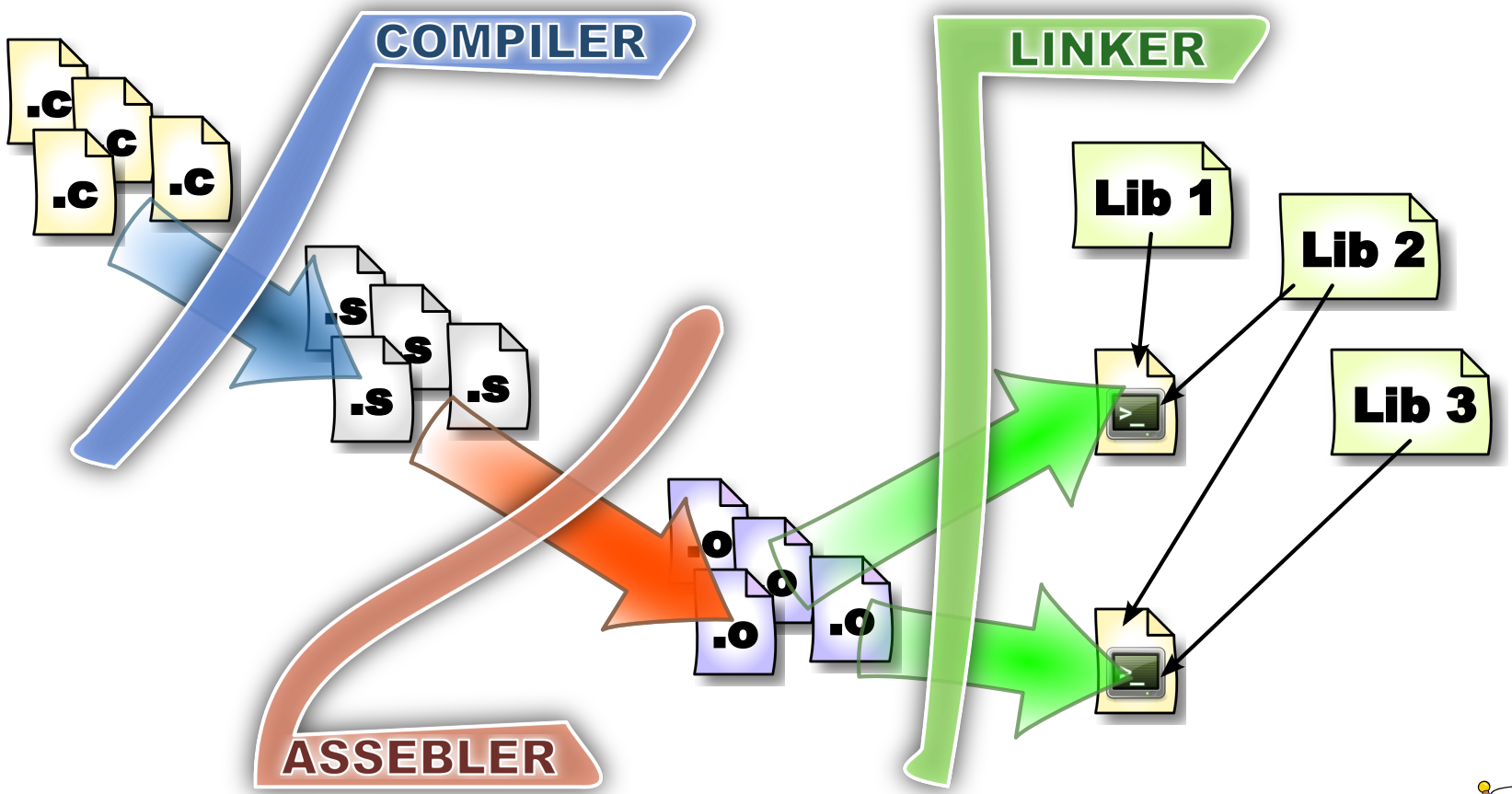
Attenzione

Non è un'operazione scontata la compilazione da un host verso un *target* molto diverso, spesso si fa uso di *tool* già pronti.



Dall'host al target

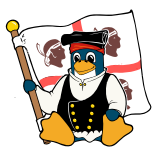
toolchain



Sviluppo multipiattaforma
- Portabilità
- Problemi correlati

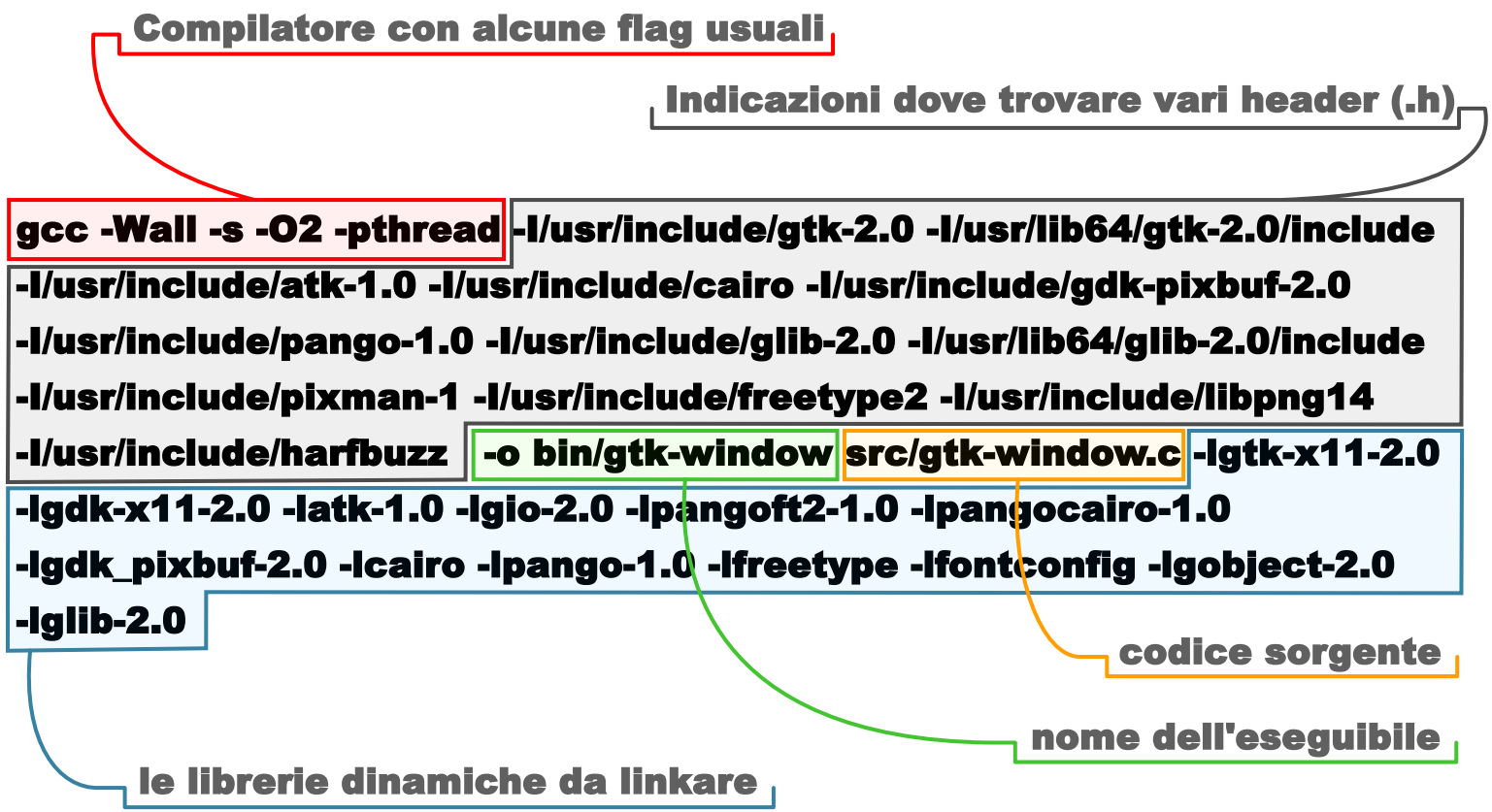
Scenari a confronto, problemi
- Analisi pre-stesura del codice
- Cross-development sul codice
- Librerie di terze parti
- Virtual Machine

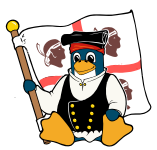
Cross-compiling
- Dall'host al target
- Creare l'eseguibile per il *target*



Creare l'eseguibile per il *target*

esempio Linux/x86_64 per lo stesso target





Creare l'eseguibile per il *target*

esempio di cross-compiling Linux/x86_64 per Windows/IA32

Compilatore con alcune flag usuali e altre utili per il cross-compiling

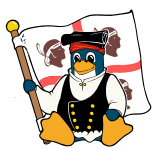
Indicazioni dove trovare vari header del TARGET

```
i386-mingw32-gcc -mwindows -mno-cygwin -mms-bitfields -s
-Wl,-subsystem,windows -Wall -O2 -Iwin32/gtk+2/include/gtk-2.0/
-Iwin32/gtk+2/include/glib-2.0/ -Iwin32/gtk+2/lib/glib-2.0/include/
-Iwin32/gtk+2/lib/gtk-2.0/include/ -Iwin32/gtk+2/include/pango-1.0/
-Iwin32/gtk+2/include/cairo/ -Iwin32/gtk+2/include/gdk-pixbuf-2.0/
-Iwin32/gtk+2/include/atk-1.0/ -o bin/gtk-window.exe src/gtk-window.c
-Lwin32/gtk+2/lib -lgtk-win32-2.0 -lglib-2.0 -latk-1.0 -lcairo
-lgdk_pixbuf-2.0 -lgio-2.0 -lgmodule-2.0 -lgobject-2.0 -lpango-1.0
-lpangocairo-1.0 -lpangowin32-1.0
```

codice sorgente

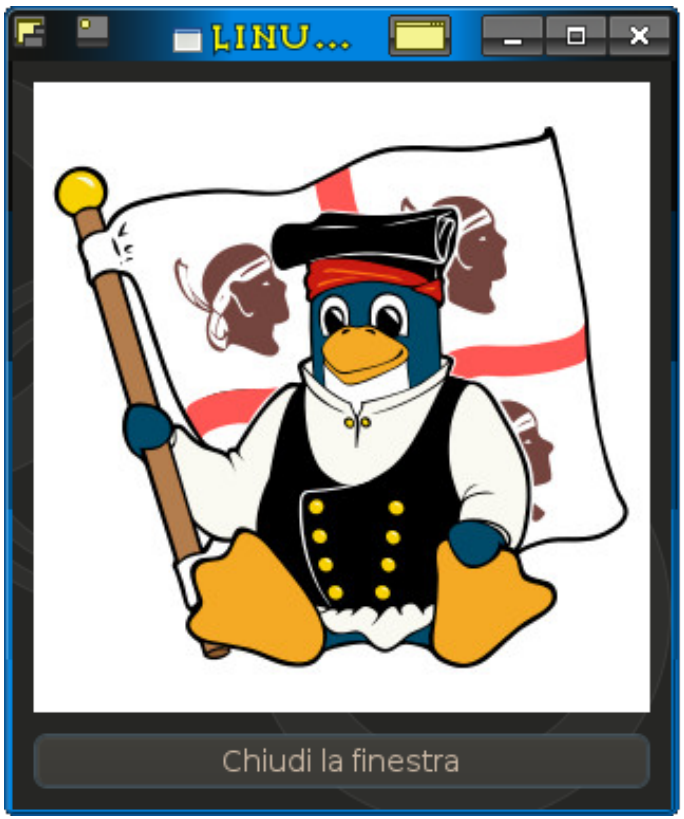
nome dell'eseguibile

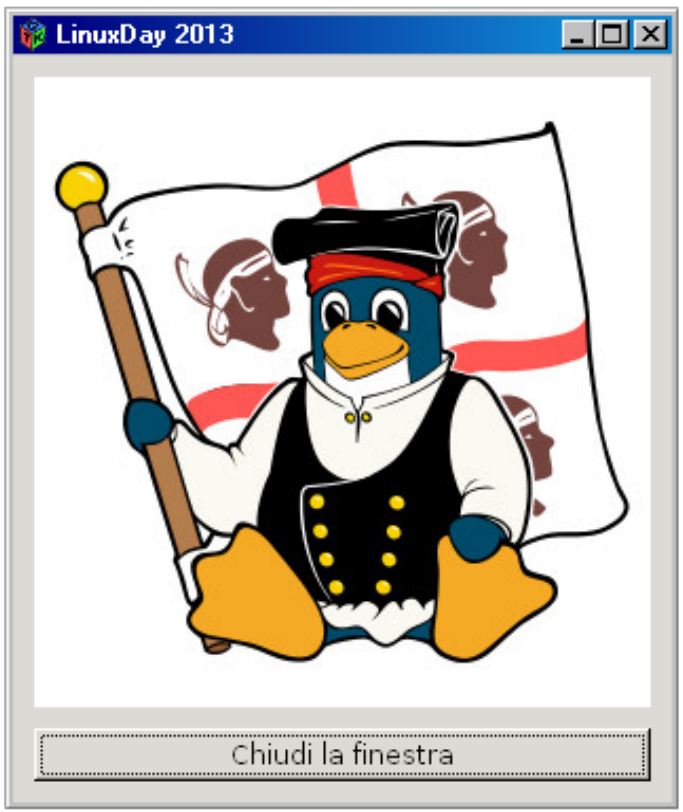
dove trovare le librerie dinamiche da linkare (TARGET)

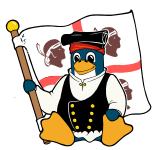


Creare l'eseguibile per il *target*

screenshot dell'applicazione in Linux/x86_64 e in Windows/IA32







Riferimenti e testi

[Wiki] - Wikipedia, the free encyclopedia

[Som] - *Ian Sommerville*, Software Engineering 8

[See] - *Peter Seeback*, A cross-compiling primer

[SGG] - *Silberschatz-Galvin-Gagne*, Operating systems concepts

FINE