Parte 15

Graphical User Interface

[V. Van Gogh – Wheat Field Under Threatening Skies, 1890]

GUI

- **Graphical User Interface**

  http://en.wikipedia.org/wiki/GUI

- Una **graphical user interface** è un tipo di interfaccia utente che permette all'utente di interagire con un programma in modi differenti rispetto alla digitazione di testo
  - Elementi grafici, icone, pulsanti, indicatori visuali, ecc.
  - Es. Desktop environment

Model-View-Controller

- **Model-View-Controller (MVC): paradigma** diffuso nello sviluppo di interfacce grafiche di sistemi software
- Basato sulla separazione dei ruoli fra i 3 componenti software:
  - **model:** rappresenta i dati e fornisce i metodi per accedere ai dati utili all'applicazione;
  - **view:** visualizza i dati contenuti nel **model** e si occupa dell'interazione con l'utente;
  - **controller:** riceve i comandi dell'utente (in genere attraverso il **view**) e li attua modificando lo stato degli altri due componenti

Struttura MVC

Lo **schema MVC** implica anche la tradizionale separazione fra:

- **Logica applicativa** ("**business-logic**") a carico del **controller** e del **model**
  - Per business-logic intendiamo l'algoritmo e le strutture dati usate per effettuare le elaborazioni di una applicazione
- **Interfaccia utente** a carico del **view**

Schema interazioni MVC (1)

- L'utente **interagisce** con l'interfaccia in qualche modo (es., preme il bottone del mouse - click)
- Il **view** notifica il **controller** dell'evento di input
- Il **controller** gestisce l'evento, spesso attraverso un **handler** o una **callback** registrata, convertendo l'evento in una azione utente corrispondente, comprensibile per il **model**
  - **Callback** e **handler** saranno per noi sinonimi, col significato di “*funzione da agganciare ad un evento*”, ossia da chiamare per gestire un evento
  - L'aggancio avviene tipicamente utilizzando un **puntatore a funzione**

Schema interazioni MVC (2)

1) Il **controller** notifica il **model** dell'azione utente, possibilmente risultante in un cambiamento nello stato stesso del **model** (es., il **controller** aggiorna lo user shopping cart)

2) Il **view** interroga il **model** per ottenere dati e generare una interfaccia utente appropriata (es., elenco del contenuto di uno shopping cart); in molte implementazioni il **view** è automaticamente avvertito dal **model** dei cambiamenti del proprio stato che richiedono un update della schermata

3) Il **view** (interfaccia utente) si (ri)mette in attesa di ulteriori interazioni, che fanno ri-iniziare il ciclo da 1)

WIMP

- Le **GUI** utilizzano una serie di elementi che costituiscono un **linguaggio visuale** per rappresentare le informazioni memorizzate su un computer ed interagire con esse
- La combinazione più comune di questi elementi nelle **GUI** si rifa al paradigma **WIMP** *("window, icon, menu, pointing device")*, che denota un modello di interazione che sfrutta gli elementi elencati

Elementi strutturali (1)

Window

- Area sul monitor che visualizza informazioni in modo indipendentemente dal resto dello schermo

Menù

- Permettono all'utente di **eseguire comandi** scegliendo da una lista di possibilità; la scelta avviene tramite un **mouse** o un altro **pointing device in una GUI** (oppure una **tastiera**)
- Mostrano esplicitamente quali comandi sono disponibili, limitando la conoscenza pregressa richiesta all'utente per usare il software

Elementi strutturali (2)

Icone

- Piccola immagine che rappresenta oggetti quali file, programmi, pagine Web, o comandi; scorciatoia per eseguire comandi, aprire documenti, ecc.

Widget (o Controlli)

- Elementi dell'interfaccia che l'utente usa per interagire col software (vedremo meglio...)

Tab

- Tipicamente un piccolo box rettangolare che contiene label testuali o icone grafiche per selezionare documenti/comandi

X Window o X11 (1)

- **X Window System (X11)**: di fatto il **gestore grafico standard** per tutti i sistemi **Unix**

http://en.wikipedia.org/wiki/X_Window_System

- **X11** fornisce l'ambiente e i componenti di base per le interfacce grafiche
  - Il disegno e lo spostamento delle finestre sullo schermo e l'interazione con il mouse e la tastiera
- **X11** non gestisce invece l'interfaccia grafica utente o lo stile grafico delle applicazioni: tutti questi aspetti sono gestiti direttamente da ogni singola applicazione

X Window o X11 (2)

Aspetti essenziali:

- Le finestre fornite dal **server X** sono solo delle aree rettangolari caratterizzate dalle proprie coordinate e dimensioni
  - Sarà poi compito dell'applicazione gestirne il contenuto sotto ogni punto di vista
- Il **server X** si occupa di fornire **eventi** alle applicazioni
  - Data una certa finestra aperta da una applicazione, il **server X** informa l'applicazione di tutti gli eventi che riguardano la finestra

Eventi

Tra gli **eventi** più importanti vi sono:

- **Click del mouse** all'interno della finestra
- **Rilascio del mouse** all'interno della finestra
- **Expose** (di una parte) **della finestra**: evento che viene generato se tale (parte della) finestra era prima coperta da un'altra finestra ed ora è di nuovo visibile

Widget

- Tutte le componenti di una interfaccia grafica sono comunemente chiamate **widget** o **controlli**
- Un **widget** in grado di contenere altri **widget** al suo interno si dice essere un **Container**

  http://en.wikipedia.org/wiki/GUI_widget
- **Widget toolkit**: un set di **widget**, fornito spesso col sistema operativo o con l'ambiente grafico, che può essere usato dai programmatori per realizzare la **GUI** di una applicazione

GTK+

- Ora possediamo tutti i concetti necessari per incominciare a costruire una nostra interfaccia
- Useremo il **widget toolkit GTK+**

  http://www.gtk.org/
- Esiste un tutorial ufficiale - **GTK+ 2.x -** accessibile nella sezione documentazione del sito

Pacchetti da installare

- Per installare gli strumenti che saranno utilizzati ed avere accesso in locale a molta della documentazione occorre installare i pacchetti:
- **GTK+ toolkit:**
  - **libgtk-3.0-dev, libgtk-3.0-doc**
- **Glib** (già visto nella lezione relativa):
  - **libglib2.0-dev, libglib-2.0-doc**
- **Glade-3** (Glade Interface Designer):
  - **glade-3, glade-doc**

Pacchetti opzionali

Per usare le vecchie funzionalita' di **libglade** (deprecato dal nuovo GtkBuilder), occorre installare:

- **glade-gnome** (applicazione **glade-gtk2**: versione di Glade che fornisce la possibilità di salvare in formato libglade)
- **libglade2-0, libglade2-dev**

Nota

- Ci interessa solo incominciare ad assimilare i concetti di base di GTK+
  - Diamo solo un'occhiata al tutorial, sarà tutto più chiaro in seguito

- Come vedremo a breve, il codice che scriveremo sarà ancora più semplice grazie all'uso di **Glade**

Documentazione di riferimento

Familiarizziamo con i **widget GTK+,** utilizzando la documentazione di riferimento (**manuale**) del **GTK+**, che troviamo:

- sul sito http://www.gtk.org/ nella scheda della **documentazione**, sezione **API**, riga **GTK**, versione **Stable**
- in locale, se installata, in

  usr/share/doc/libgtk-3-doc/gtk3/index.html

Galleria dei Widget

- Aprire il capitolo **Widget Gallery**
  - IV. GTK+ Widgets and Objects
- Guardare come funzionano almeno i seguenti widget: **window, label, button, toggle button, check button, radio button, entry, (multiline) textview, menu bar**
  - Per vedere il nome di ciascun **widget** basta passarci sopra col mouse
  - Per avere informazioni su ciascun **widget** cliccarci sopra e si viene rimandati alla relativa pagina nel manuale di riferimento

Formato pagine manuale (1)

- **Nome del widget descritto**, sua descrizione breve, nonché immagine se disponibile
- **Synopsis**
  - Fondamentalmente l'interfaccia del widget
- **Object Hierarchy**
  - La gerarchia delle classi, ossia tutte le classi da cui deriva l'oggetto a partire da quella base
- **Implemented Interface** (per ora non ci interessa)

Formato pagine manuale (2)

- **Properties e Style Properties**
  - Sono chiamate **Properties** tutti i campi di un **widget** che ne caratterizzano le proprietà
- **Signals**
  - La vedremo fra breve
- **Description**
  - Finalmente una descrizione lunga del **widget**

Formato pagine manuale (3)

- **Details**
  - Descrizione dettagliata dell'interfaccia del modulo (**widget**)
- **Property Details / Style Property Details**
  - Descrizione dettagliata di tutte le proprietà elencate nella sezione **Properties**
- **Signal Details**
  - La vedremo fra breve
- **See Also**
  - Riferimento a pagine correlate

Eventi, segnali e handler (1)

/usr/share/doc/libgtk2.0-doc/tutorial/book1.html

Diamo uno sguardo al capitolo **Theory of Signals and Callback**

- **GTK** è un **toolkit event-driven**
- Significa che resterà **dormiente** (in una fase di **sleep - gtk_main()**) fino a quando non avverrà un **evento**
- Al verificarsi di un evento il **controllo** sarà passato alla **funzione appropriata (handler o callback)**

Eventi, segnali e handler (2)

- Riassumendo, prima di tutto un **widget** riceve un **evento**
- L'evento parte dal **server X,** che lo trasmette fino ad arrivare al **widget GTK+** visualizzato attraverso quella **finestra del server X**
- Il **widget** riceve l'evento ed **emette in conseguenza un segnale**
- Mediante la funzione **g_signal_connect** agganciamo all'evento tutti gli **handler** (o **callback**) che vogliamo

Eventi, segnali e handler (3)

- L'emissione del segnale è una “metafora” per descrivere il meccanismo di **invocazione di una serie di funzioni “agganciate”**
- <u>Esempio</u>:
- Se si preme e si rilascia il bottone sinistro del mouse mentre il puntatore è su un **widget bottone**, il bottone riceve l'**evento clicked**
- In questo caso il bottone emette il **segnale clicked**, *che comporta l'esecuzione di tutte le funzioni che abbiamo agganciato a tale segnale*

Eventi, segnali e handler (4)

- Come si è visto un **handler** è una funzione da eseguire
- Come è fatta la **dichiarazione** di tale funzione?
- L'esatto **prototipo** da usare varia da segnale a segnale, come stiamo per vedere...

Sezioni relative ai segnali

Come visto, ciascuna pagina di manuale che descriva un **widget** (*Widget Gallery*) contiene anche le seguenti **due sezioni**

- **Signals**

  La lista di tutti i segnali che possono essere emessi da un **widget**
- **Signal details**

  Il prototipo che deve avere **l'handler** di ciascun segnale, ed il significato dei suoi parametri di ingresso e del valore di ritorno

Dichiarazione di un handler

Riassumendo, per agganciare un **handler** ad un segnale dovremmo

- Definire la funzione **handler,** dandole il nome che preferiamo, ma rispettando la **dichiarazione** riportata nella sezione **Signal Details** della pagina di manuale
- Agganciare la funzione **handler** mediante una **g_signal_connect**
  - A breve vedremo come questa operazione può essere effettuata in modo più semplice con **Glade**

gtk_main

- Dal punto di vista dell'interfaccia grafica, un'**applicazione GTK+** non fa altro che inizializzare la libreria, agganciare tutti gli **handler** ed invocare la funzione **gtk_main()**
- La funzione **gtk_main()** esegue un **ciclo infinito** in cui aspetta il prossimo evento dal **server X**, lo passa al **widget** interessato, esegue gli **handler** agganciati ai segnali emessi dal **widget**, poi torna ad aspettare il prossimo evento
- **Schema event-driven**

Esempio

- Aprire il file *helloworld.cc*
- Esempio di programma che apre una finestra dal titolo “Hello, world!”, contenente una label “Hello, world!!!”
- Compilare con:

  g++ helloworld.cc `pkg-config --cflags --libs gtk+-3.0`

Glade

- Inserire i **widget** ed agganciare gli **handler** a mano è ripetitivo, noioso e fonte di errori
  - Approccio ormai **sconsigliato**
  - Si utilizzano **strumenti visivi** per il disegno di interfacce
- Per realizzare il nostro primo programma passiamo ad utilizzare **Glade (Glade-3)**

  http://glade.gnome.org
- **RAD tool (Rapid Application Development)**

Formato XML

Glade genera una decrizione dell'interfaccia sviluppata in formato XML (file .glade)

- **XML** è un semplice formato testuale
- Formato di uso generale, molto utile per **descrivere** il contenuto di un documento o in generale di un oggetto
- Permette di esprimere in modo semplice l'elenco dei **widget** che ci sono nella nostra interfaccia e le loro relazioni (es., quali **widget** sono contenuti in un certo **widget** contenitore)

Caricamento interfaccia

- L'interfaccia grafica generata con **Glade** deve essere **caricata** dentro il nostro programma
- Il modo più semplice per caricare l'interfaccia è usare funzioni di librerie apposite:
  - **GtkBuilder** (oggetto integrato in GTK), oppure
  - **libglade** (libreria addizionale, in disuso)
- **L'interfaccia grafica caricata** dentro il nostro programma viene fatta apparire sullo schermo

GtkBuilder vs libglade

- Le ultime versioni di **Glade** non permettono piu il salvataggio dell'interfaccia in formato compatibile con libglade
  - deprecata a favore di GtkBuilder
- Meglio adottare GtkBuilder (altrimenti occorre usare il vecchio **glade-gnome**: comando **glade-gtk2**)

Sviluppo interfaccia con Glade

I compiti distinti da svolgere per realizzare un programma grafico utilizzando **Glade** sono:

1. **Disegnare l'interfaccia (con Glade)**
2. Scrivere il programma con il nostro **linguaggio di programmazione**; il programma dovrà:
   - caricare l'interfaccia dal file **xml** (**.glade**) e farla apparire;
   - contenere tutti gli **handler** che vogliamo usare

Come vedremo, per ciascun **widget** è possibile agganciare gli **handler** ai segnali del **widget** stesso direttamente da **Glade**

GtkBuilder

- GtkBuilder è un oggetto ausiliario che effettua il parsing di interfacce utente specificate in formato testuale (XML) creando gli oggetti descritti

- Creazione di un nuovo oggetto GtkBuilder

```
GtkBuilder *builder;
builder = gtk_builder_new();
```

- https://developer.gnome.org/gtk3/stable/GtkBuilder.html

Caricamento interfaccia

- Partendo da un interfaccia composta con Glade, è possibile caricarla nel nostro programma attraverso GtkBuilder chiamando **gtk_builder_add_from_file ()**
- Es.:

```
gtk_builder_add_from_file(builder,
     "primaGUI.glade",NULL);
```

Aggancio degli handler

- La funzione **gtk_builder_connect_signals()** aggancia tutti gli **handler** che sono definiti nell'interfaccia creata con **Glade-3** (come vedremo a breve)
- Tale funzione deve però poter accedere ai nostri **handler** per svolgere tale compito
- Quindi, in quanto agli **handler**, stavolta si va nel verso contrario: non sono le funzioni scritte nel nostro programma ad accedere alle funzioni di libreria, ma il contrario

Collegamento dinamico

- Di **default**, quanto all'accesso alle funzioni contenute nel programma che si compila, il compilatore **g++** assume collegamento statico
  - Ma le librerie sono invece collegate dinamicamente!!
- Normalmente la **tabella dinamica dei simboli** contiene solo i simboli usati da oggetti dinamici
  - Abbiamo bisogno di **inserire i nostri simboli nella tabella dinamica!**
- L'opzione **-export-dynamic** fa questo: aggiunge tutti i simboli alla tabella dinamica dei simboli

Opzione -export-dynamic

- Per visualizzare la dynamic symbol table di un file oggetto possiamo usare **objdump** con l'opzione **-dynamic-syms** (oppure **-T**)
  - Con -t vediamo invece la symbol table normale (non quella dinamica)
- Provare a vedere come cambia la **dynamic symbol table** compilando un programma con o senza l'opzione **-export-dynamic**

Il comando pkg-config (1)

- Per capire cosa fa il comando **pkg-config,** invocarlo prima da solo con i seguenti argomenti:

  pkg-config --cflags gtk+-3.0

- L'output è formato da una serie di opzioni **-I (i maiuscola)** seguite da nomi di directory
- Si tratta dell'elenco delle **directory** in cui il compilatore deve cercare gli **header file** se si vuole utilizzare la **libreria gtk+-3.0**

  pkg-config ritorna metainformazioni sulle librerie installate

Il comando pkg-config (2)

- Invochiamo ora il comando con l'altra opzione:

  pkg-config --libs gtk+-3.0

- Questa volta l'output è la lista di opzioni da passare al **linker** per informarlo di tutti i file di libreria a cui deve essere collegato il programma per utilizzare la **libreria gtk+**
- Se invochiamo infine il comando con entrambe le opzioni **--cflags e --libs** otteniamo entrambe le sequenze di opzioni per il compilatore **g++**

Il comando pkg-config (3)

- Dall'esempio si capisce che il comando **pkg-config** ci permette di evitare di dover passare a mano tutte queste opzioni al compilatore
- **Uso degli apici inversi `**
  - Per sostituire una parte di una riga di comando con i caratteri che sarebbero stampati (output) da un altro comando si racchiude tale comando tra apici inversi `
  - Sulle tastiere italiane l'apice inverso si ottiene di norma con **AltGr + '**

Esempi

- Per compilare un programma con interfaccia grafica scritto in **C++** con un solo comando:

  ```
  g++ -Wall -export-dynamic prog.cc `pkg-config --cflags --libs gtk+-3.0`
  ```

- Invece, se si separano le due fasi di traduzione e collegamento:

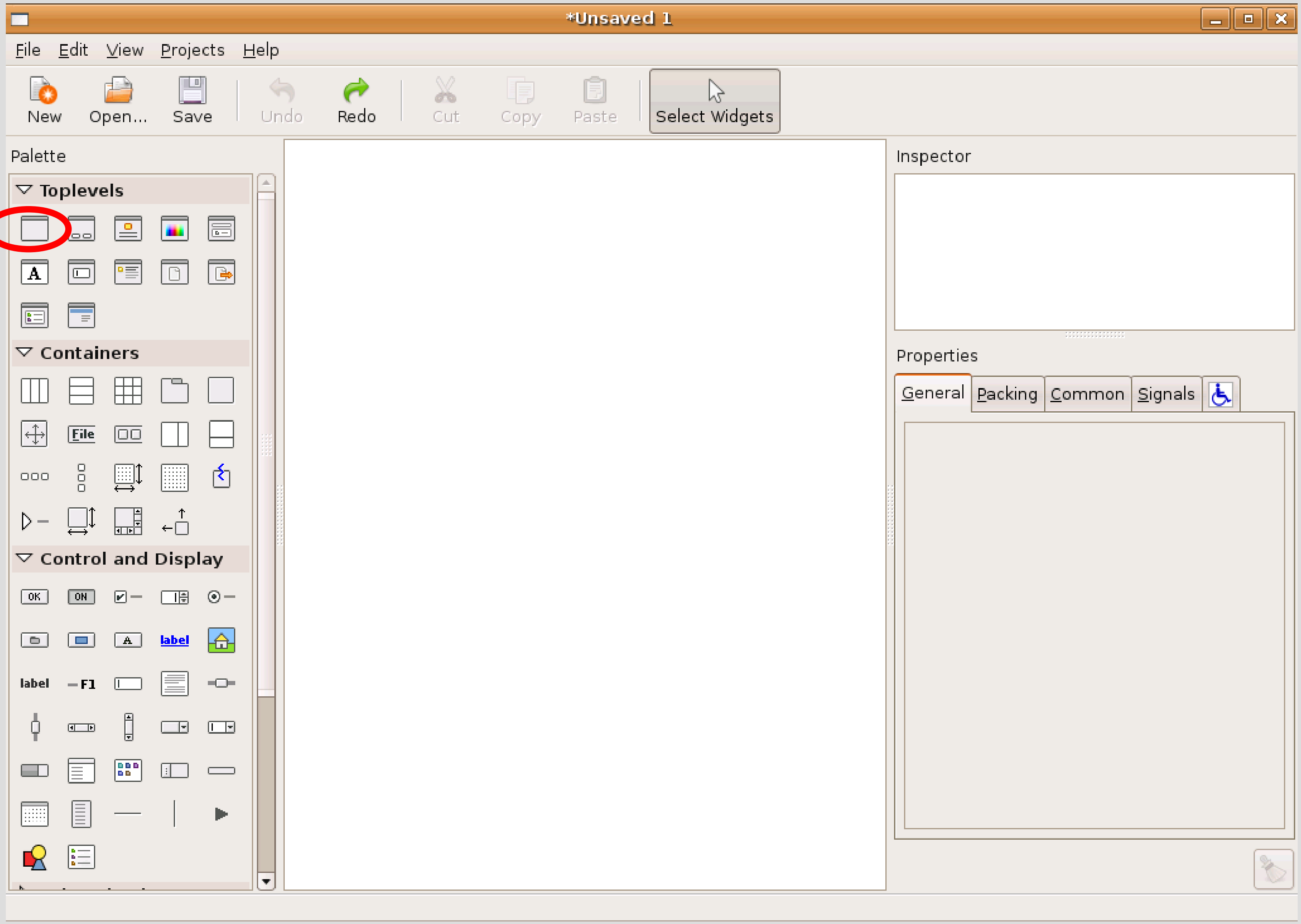
  ```
  g++ -Wall -export-dynamic -c prog.cc `pkg-config --cflags gtk+-3.0`
  ```

  ```
  g++ -Wall -export-dynamic prog.o `pkg-config --libs gtk+-3.0`
  ```

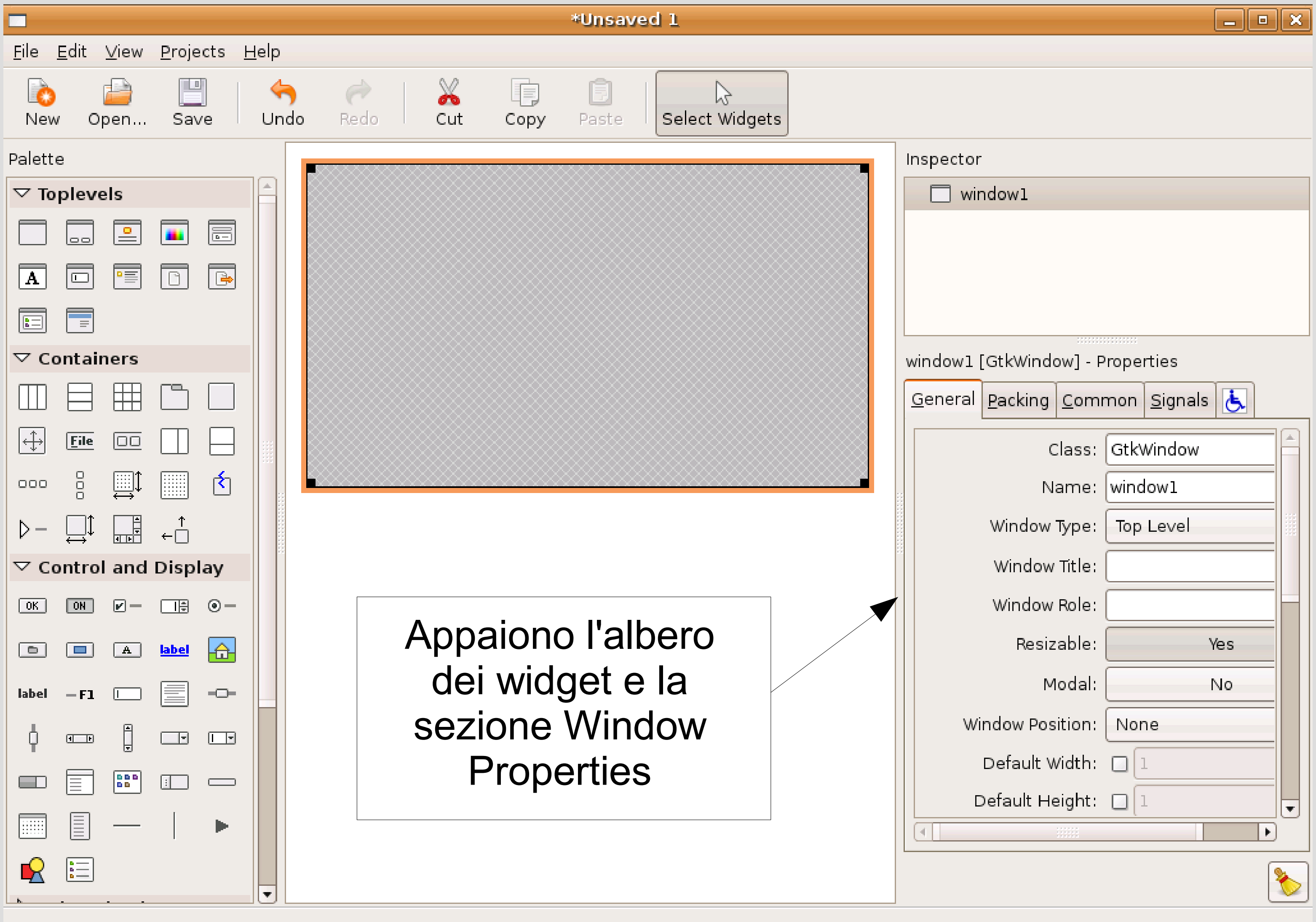
Inizio

- Ora possediamo tutte le informazioni che ci servono per scrivere il nostro primo programma con **Glade-3** e **GTK+**
- *Proviamo a scrivere un programma che faccia semplicemente apparire una finestra*
- Cominciamo dal disegnare l'interfaccia con **Glade-3**
  - **Comando glade**

Selezione finestra (window)



Dopo la selezione



Appaiono l'albero dei widget e la sezione Window Properties

Salvataggio

- Selezioniamo la voce di menù **Save As,** scegliamo la cartella in cui salvare il file **xml** contenente l'interfaccia, scegliamo il nome del file e salviamolo
- Non occorre aggiungere suffissi al nome del file
  - **Glade-3** aggiunge automaticamente il suffisso **.glade**

Nota

- Apriamo con un **editor di testo** il file generato
- Anche se non conosciamo il formato **xml** ci accorgiamo che il senso del contenuto del file è grossomodo interpretabile (a parte i dettagli tecnici)
- Uno dei vantaggi di questo formato è che per fare modifiche ripetitive, come ad esempio cambiare nome ad un **handler** e quindi modificare tutte le occorrenze di tale nome nel file **xml**, possiamo sfruttare efficacemente le funzioni di un editor di testo

Codice

- Scriviamo ora il codice che
  - Inizializzi gtk e crei il builder
  - Carichi l'interfaccia creata
  - Connetta tutti i segnali
  - Inizi il ciclo di gestione degli eventi

- Soluzione in *primo_prog_solomain.cc*

Compilazione ed esecuzione

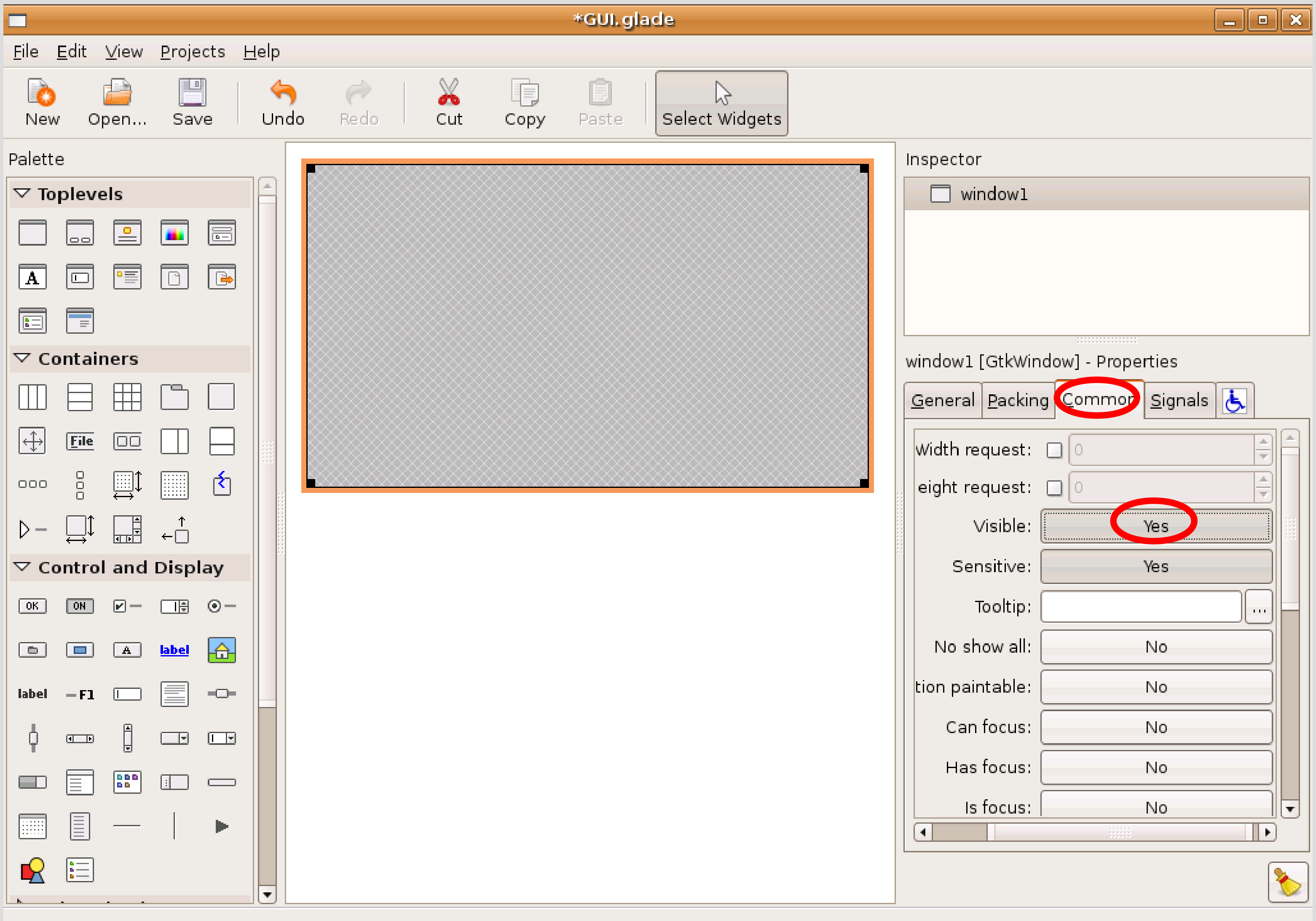
- **Compiliamo** quindi con la riga di comando vista nelle precedenti slide
- Una volta corretti tutti gli eventuali errori, eseguiamo
- Cosa appare?

Proprietà visible

Probabilmente nulla, per il seguente motivo

- Tutti i **widget** hanno una **proprietà visible** di **tipo booleano**
- Il **widget** appare se e solo se la proprietà visible ha valore vero
- Per settare a vero o falso tale proprietà per un **widget** *e tutti i suoi figli*, e quindi far apparire o sparire il **widget** con tutti i suoi figli durante l'esecuzione, sono previste opportune *funzioni*
- Ma si può anche settare subito manualmente la proprietà a vero tramite **Glade** ...

Proprietà visible



Salviamo e rieseguiamo

- Salviamo quindi la nuova versione dell'interfaccia e rieseguiamo il programma
- E' **necessario ricompilare** anche il sorgente?
- **No**, perché il programma semplicemente si ritrova la nuova versione dell'interfaccia quando la carica con **gtk_builder_add_from_file()**
  - Questo è uno dei vantaggi di **Glade-3!**
- A questo punto dovrebbe apparire correttamente la finestra e siamo pronti per il prossimo passo

Window manager

- Come si vede il **window manager** ha avuto cura di aggiungere la barra superiore ed i bottoni
- Proviamo quindi a chiudere la finestra con il bottone a X in alto
- La finestra si chiude
- *Ma il programma?*

Eliminazione di una finestra

- Il programma non si chiude perché il **window manager** semplicemente informa il **widget window** che lo farà sparire, e questo fa sì che il **widget** emetta il segnale **delete_event**
- Se nessun **handler** è agganciato a questo segnale non succede niente altro
  - Non ci resta che premere **Ctrl+C** dal terminale per terminare il programma
- Se invece si aggancia un **handler** al segnale, tale **handler** deve ritornare vero o falso
  - Se ritorna vero si esegue **l'handler e basta**

Segnale destroy

- Se invece **l'handler** ritorna falso, terminata la sua esecuzione il **widget** emette anche il **segnale destroy**
- Di nuovo il meccanismo è lo stesso: se non si è agganciato nessun **handler** al **destroy** non succede nient'altro, altrimenti parte **l'handler** agganciato al **segnale destroy**
- C'è quindi un **meccanismo in due fasi successive** associato alla pressione del tasto di **eliminazione della finestra**

Motivazione (1)

- Questo meccanismo in due tempi permette all'applicazione di reagire al tentativo di chiusura errata
- Il **primo handler** può chiedere all'utente se è sicuro o può far compiere altre azioni prima della chiusura
- Se poi è veramente il caso di chiudere, l'**handler** ritorna **FALSE** e viene emesso il segnale destroy, altrimenti ritorna **TRUE** e l'applicazione si rimette in attesa del prossimo evento

Motivazione (2)

- In questo modo al segnale **destroy**, tipicamente generato in conseguenza di eventi '**definitivi**', si può agganciare un **handler** che per esempio chiuda definitivamente il programma
- Proviamo quindi a partire aggiungendo al nostro programma un handler per il **delete_event** che ritorni **vero**

Booleani in GTK+

- GTK+ è scritto in **C** quindi non ha tipo booleano
  - Negli **header file** di **GTK+** sono definite due costanti intere, **TRUE** e **FALSE**, che contengono, rispettivamente, il valore intero che corrisponde a vero e quello che corrisponde a falso in linguaggio **C/C++**
- Anche se di fatto è un intero, il tipo di tali costanti è chiamato **gboolean** per leggibilità
- Anche se il nostro programma è scritto in **C++**, per compatibilità ci può convenire comunque utilizzare tali costanti piuttosto che quelle booleane

Handler (1)

- Come deve essere fatta l'intestazione dell'**handler** del segnale **delete_event**?
- Andiamo nella sezione **Signal Details** della pagina del manuale di riferimento di **Window**

  file:///usr/share/doc/libgtk-3-doc/gtk3/GtkWindow.html

- Questo segnale è caratteristico di ogni **widget:** un **widget** di tipo **window** lo *eredita* per il fatto che è un tipo di **widget**
- La dichiarazione da usare per **l'handler** del segnale **delete_event è nella pagina dedicata al Widget generico *(GtkWidget)***

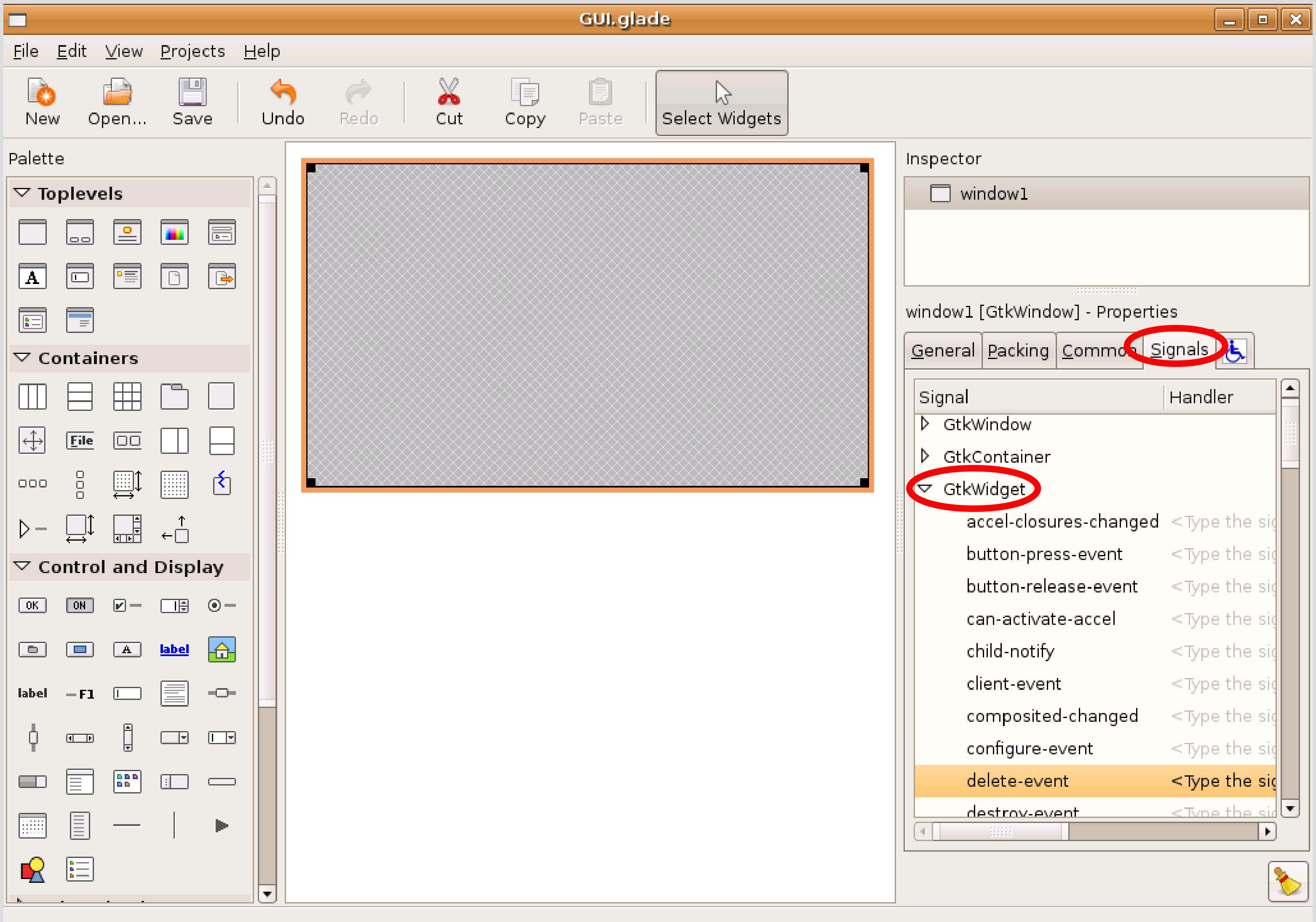
Handler (2)

- Definiamo pertanto nel nostro programma una funzione che rispetti tale dichiarazione
- Per far prima, *incolliamo la dichiarazione* nel nostro codice e poi modifichiamo quello che c'è da modificare
- Chiamiamo la funzione **handler_delete_event**
- Facciamo ritornare **TRUE** alla nostra funzione
- Facciamole semplicemente stampare un messaggio su **stdout**
  - Per ora non usiamo nessun parametro formale

Interfaccia (1)

- Scritta la funzione, ricompiliamo il programma
  - Se ri-eseguiamo non è cambiato nulla
- Dobbiamo procedere ad agganciare l'**handler** al segnale mediante **Glade**
- Andiamo nella cartella **signal**, apriamo i segnali in **GtkWidget** e cerchiamo **delete_event**, come mostrato nella prossima slide

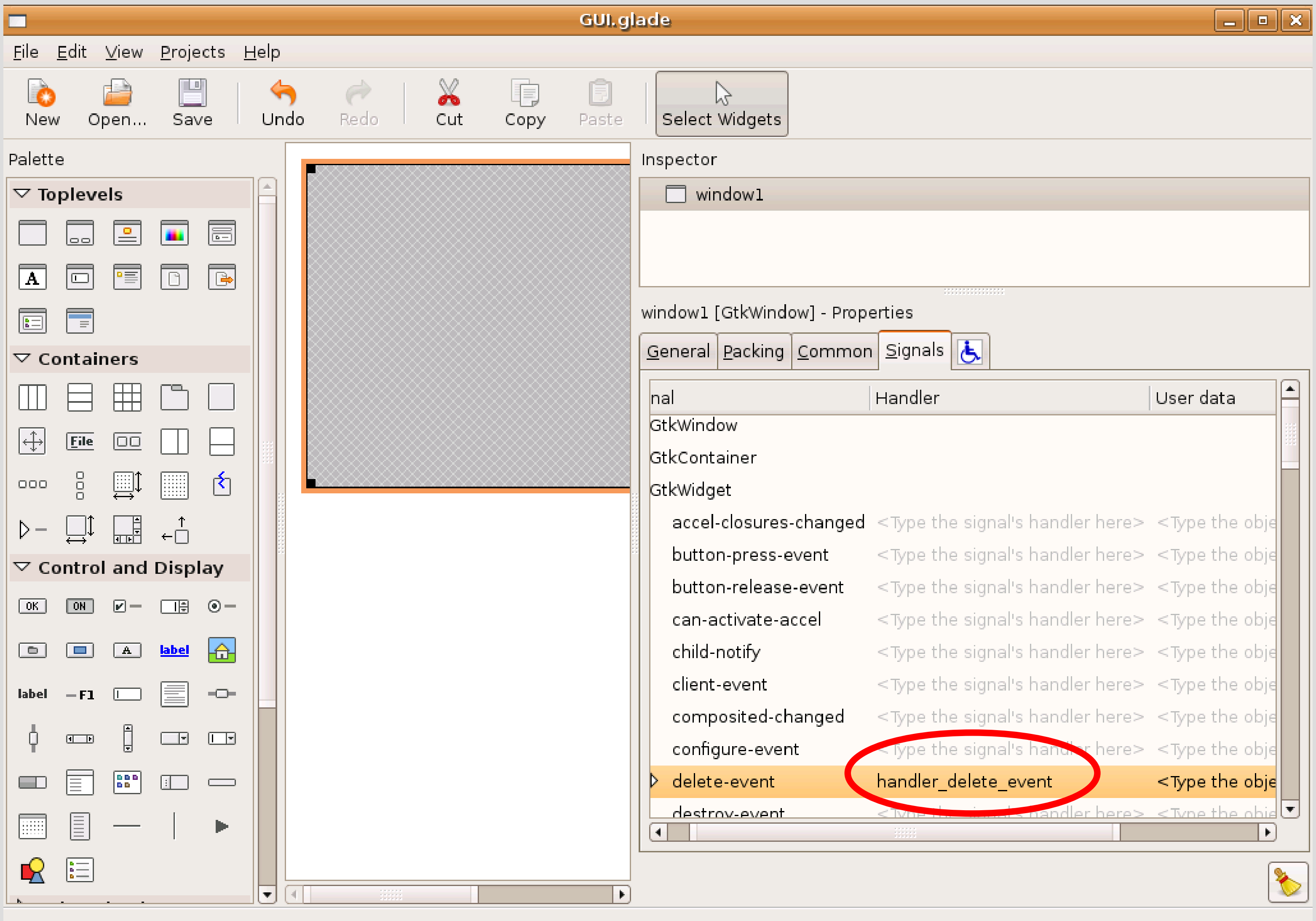
Interfaccia (2)



Interfaccia (3)

- Scriviamo il nome dell'**handler** nel campo subito a destra del nome del segnale **delete_event** e premiamo invio, come mostrato nella prossima slide, e salviamo l'interfaccia

Interfaccia (4)



Accesso da GtkBuilder (1)

- Andiamo quindi ad **eseguire** il programma
- Alla pressione del tasto di eliminazione della finestra è molto probabile che su stdout appaia un messaggio del tipo:

  ... Gtk-WARNING **: Could not find signal handler 'handler_delete_event'
- Provare a capire il motivo riflettendo sul fatto che **GtkBuilder** è scritta in **C** e non in **C++** ...

Accesso da GtkBuilder (2)

- Come visto in precedenti lezioni, c'è incompatibilità tra i simboli del file oggetto generato dalla compilazione del nostro programma in **C++** ed i simboli che si aspetta **GtkBuilder**, che è scritto in **C**
- Per far vedere a **GtkBuilder** i simboli corretti dobbiamo definire la nostra funzione come **extern "C"**
  - Aggiungere questa dichiarazione, ricompilare e rieseguire
- Alla pressione del bottone di eliminazione dovrebbe apparire il nostro messaggio di stampa

Chiusura applicazione

- Per completare il nostro semplice esempio, inseriamo il codice per la terminazione del programma alla pressione del bottone di chiusura
- Abbiamo **due possibilità**:
    1) Scrivere l'**handler** per il segnale **destroy** e far tornare **FALSE** a **handler_delete_event**
    2) Inserire un'istruzione di terminazione del programma direttamente nell'**handler** del segnale **delete_event**
- Per brevità optiamo per la seconda soluzione

Uscita dal main loop

- Per chiudere un'applicazione **GTK+** in modo pulito, possiamo invocare la funzione **gtk_main_quit()**, che fa uscire dal ciclo principale, ossia dalla funzione **gtk_main()**
- Di conseguenza sarà eseguita l'istruzione successiva all'invocazione della funzione **gtk_main()**
  - Nel nostro esempio l'invocazione di **gtk_main()** è seguita solo dall'istruzione **return**, per cui il programma termina immediatamente a seguito dell'invocazione di **gtk_main_quit()**

Completamento esempio

- Inseriamo l'invocazione della funzione **gtk_main_quit()** nell'**handler** del segnale **delete_event**
- Ricompiliamo ed eseguiamo
- Se tutto è andato bene, abbiamo ora le basi per provare a realizzare una prima **interfaccia grafica** con menù, bottoni, text view e finestre di dialogo
- Soluzione completa in *primo_prog/GtkBuilder/primo_prog.cc*