C++ and GNOME

GUADEC 2008
Istanbul, Turkey
Tuesday July 8th 2008
hfiguiere@novell.com

Novell®

Why using C++?

We don't smoke the crack pipe...

Why using C++

- Need of something higher level than C
- Need to use lot of C code
- Need something fast
- Need and “industry standard”

Higher level than C

- Strong typing
- Object Oriented
- Generic programing
- Comes with standard library with containers

Reuse C code

- C code is 99% compatible with C++
  - Changes keep it compatible with plain C
- Can call C code implicitely
- Make C code more robust if compiled with C++ compiler
  - Stricter typing

C compatibility

- type casting is neccessary. Pointers, enum, etc.
- Some keywords added conflict with potential symbols: private, public, template, etc.
- C vs C++ linkage.
  - Different name mangling
  - Make sure function called from C are surrounded by extern “C” { }
- ...

Speed

- C++ is as fast as C.
  - C code compiled as C++
- Template provide generic programing without speed penalty.
  - Most of the work is done at compile time
- Standard library optimised for speed.
  - STL hard to beat
- Can still use very low level C.

Libraries

- Standard Library, aka STL
  - Part of the ISO standard
  - Provide containers and other useful code
- Boost
  - A “boost to the STL”
  - Written to be compatible with STL
  - Part will be integrated in the Standard Library for the C++0x ISO stanard
- Gtkmm

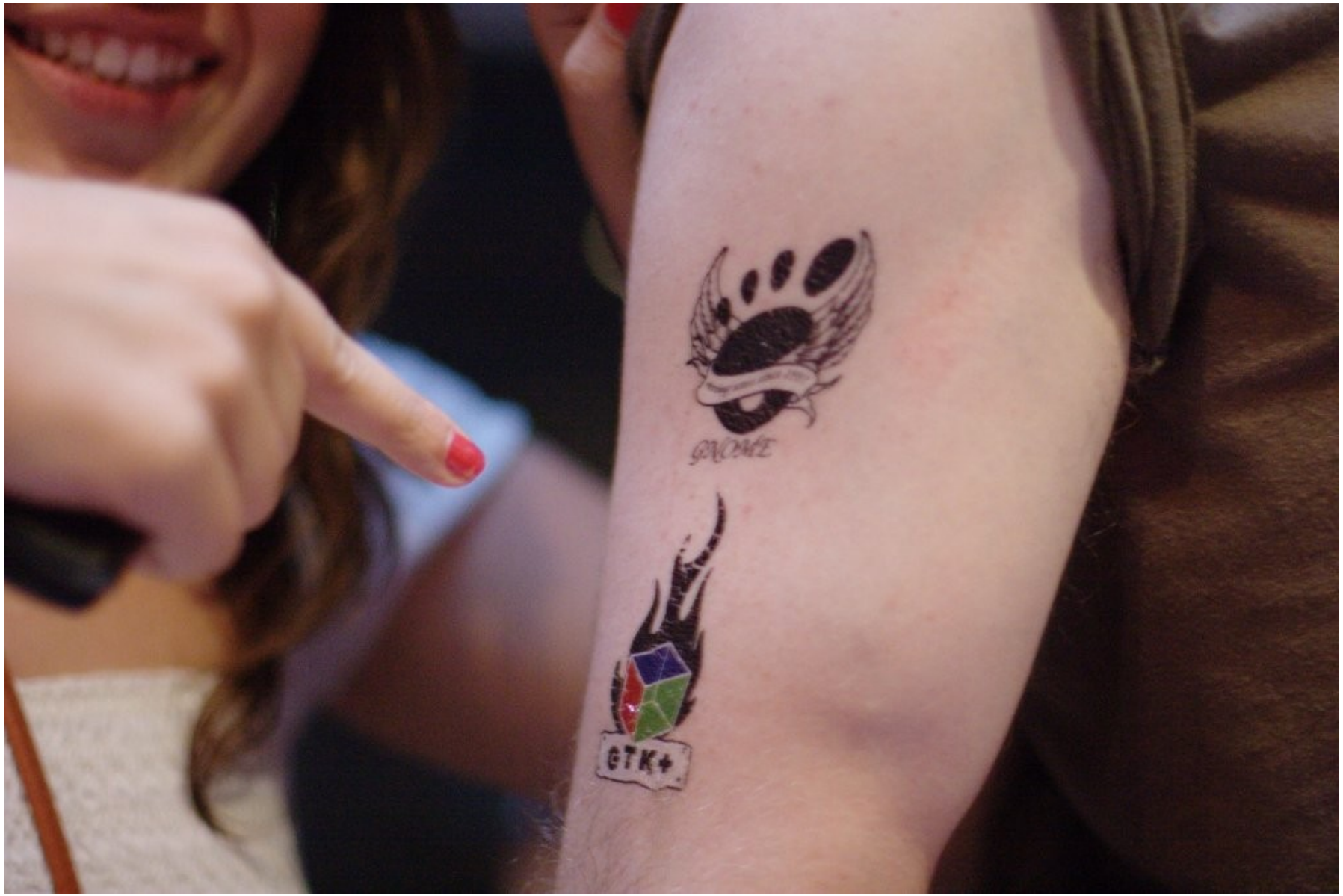
What a choice !

So many libraries!

Which one?

- STL or Standard Libray
  - Prefer it to Glib for containers
- Boost
  - Smart pointers
  - Utilities
- Gtkmm
  - Build your UI with it.

Because we love GNOME



Gtk+ and C++

- You can use Gtk+ in C++ directly
  - Example
    - Mozilla
    - OpenOffice.org
    - AbiWord
  - Still a pain to write new widgets: GObject

Gtkmm

- Gtkmm are the C++ “bindings”
  - Wrap GObject in a C++ friendly fashion
  - Type safe signals
  - Can subclass a GObject directly in C++
    - easier
  - Can still mix plain GObject code
  - Can also use these C++ objects from C
    - Although with a little cheating
- Gtkmm designated a family of API

Example 1: unwrapping

```
void function()
{
  Gtk::IconView *librarylistview;
  Gtk::CellRendererPixbuf libcell;
   // do something
  // ....
  GtkCellLayout *cl = GTK_CELL_LAYOUT(librarylistview.gobj());

  gtk_cell_layout_pack_start(cl,
                             GTK_CELL_RENDERER(libcell->gobj()),
                             FALSE);
  gtk_cell_layout_add_attribute(cl,

  GTK_CELL_RENDERER(libcell->gobj()),
                   "pixbuf",
                   m_model->columns().m_pix.index());
  gtk_cell_layout_add_attribute(cl,

  GTK_CELL_RENDERER(libcell->gobj()),
                   "libfile",
                   m_model->columns().m_libfile.index());
}
```

Example 2: subclassing

```
class LibraryCellRenderer
  : public Gtk::CellRendererPixbuf
{
public:
  LibraryCellRenderer();

  virtual void     get_size_vfunc (Gtk::Widget& widget,
                                   const Gdk::Rectangle* cell_area,
                                   int* x_offset, int* y_offset,
                                   int* width, int* height) const;
  virtual void     render_vfunc (const Glib::RefPtr<Gdk::Drawable>& window,
                                 Gtk::Widget& widget,
                                 const Gdk::Rectangle& background_area,
                                 const Gdk::Rectangle& cell_area,
                                 const Gdk::Rectangle& expose_area,

  Gtk::CellRendererState flags);

  Glib::PropertyProxy_ReadOnly<db::LibFile::Ptr>     property_libfile() const;
  Glib::PropertyProxy<db::LibFile::Ptr>      property_libfile();

private:
  Glib::Property<db::LibFile::Ptr>    m_libfileproperty;
};
```

Properties

Resource management

- C++ has advantage over C for resource management
- Resource acquisition is initialisation (RAII)
  - Constructed entering the scope
  - Destructed exiting the scope

Smart pointer

- Using RIIA we can write “smart pointer
  - A pointer that will free itself when no longer needed
  - Predictible garbage collection
- Standard library
- Boost
- Glibmm

Standard library: std::auto_ptr<>

- Very primitive:
  - Transfer pointer ownership on copy
  - Release memory on destruction.
- Avoid using. Prefer boost.

Boost.SmartPtr

- Boost provide a set of Smart pointer classes
  - shared_ptr: shared pointer
  - scoped_ptr: a non copyable pointer to free on exit
  - weak_ptr: a pointer that can be freed anytime
    - A precondtion check make it safe.
  - intrusive_ptr: for when the “resource” have it own management and ref counting.
- Is in TR1 for inclusion in the standard library

Glibmm

- Glibmm has Glib::RefPtr<>
  - A kind of intrusive pointer
  - Perform g_object_ref() and g_object_unref()
  - Used exclusively within Glibmm and Gtkmm.

Pointer: Example in C

```
void function()
{
    MyStruct * s = (MyStruct*)malloc(sizeof(MyStruct));
    if(condition) {
            /* do something */
            free(s);
            return;
    }
    /* do something else */
    free(s);
}
```

Smart pointer: Example in C++

```
void function()
{
   boost::scoped_ptr<MyStruct> s(new MyStruct));
   if(condition) {
            /* do something */
            return;
   }
   /* do something else */
}
```

Glib::RefPtr: Example

```
void function()
{
   Glib::RefPtr<Gdk::Pixbuf> pixbuf =
   Gdk::pixbuf::create(Gdk::COLORSPACE_RGB,
                                                false, 8, 100, 100);
   if(condition) {
            /* do something */
            return;
   }
   /* do something else */
}
```