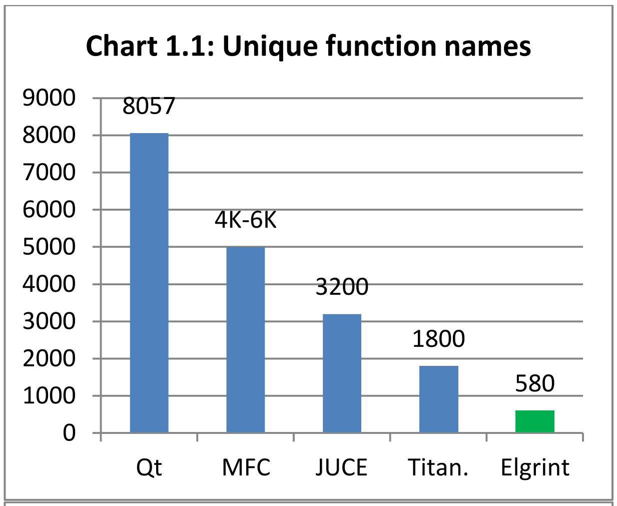
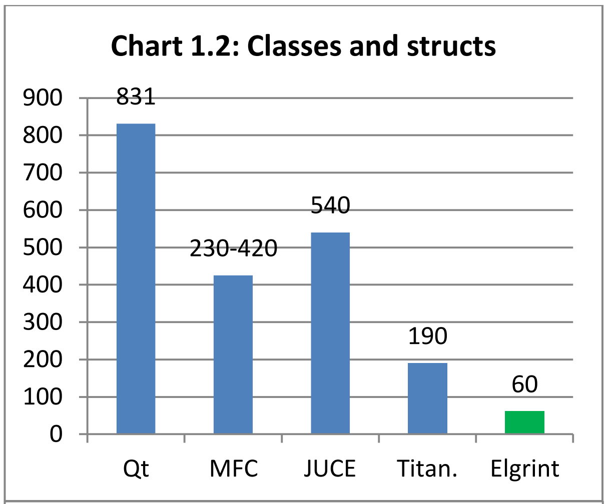
The Seven Advantages of Elgrint (Easily-learned graphic interface toolkit)

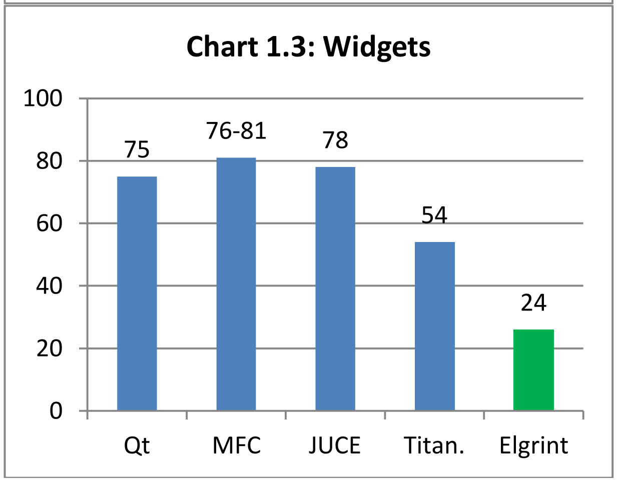
By Dimitry Rotstein, April 2014

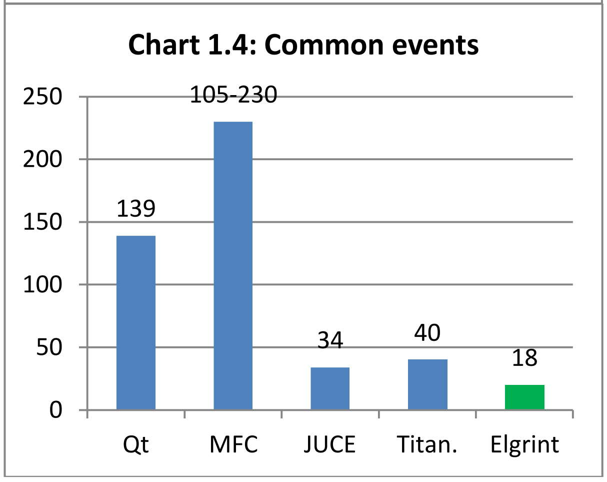
(1). Elgrint saves learning time, effort, and cost.

The following charts provide a comparison between Elgrint interface and the interfaces of the most prolific and/or relevant commercial toolkits: Nokia/Digia's **Qt** (C++), Microsoft's **MFC** (C++) – may it rest in piece, Raw Materials's **JUCE** (C++), and Appcelerator's **Titanium** (JavaScript). Toolkits are compared in terms of the number of interface elements: functions, classes (structural elements), widgets (graphic elements), and events (logical elements):









Note: Despite such incredible compactness, Elgrint contains more than **90%** of the same functionality (the remaining 10% will be added in the next versions without noticeably increasing the interface size).

Why does the interface size matter?

A more compact interface is easier to learn, use, implement, develop, support, document, and so on. The cost reduction is often linear. For example, learning Elgrint in depth should take 13-14 times less than learning Qt, say, a week instead of three months.
This saves a lot of effort for developers (it's like walking up two floors instead of 28 floors) and a lot of money for their employers (one week salary instead of 3 months salary). In fact, it takes only **a few hours** to learn all the essential parts of Elgrint (see http://www.miranor.com/elgrint/reference/guide.php)

(2). Elgrint saves development time and cost.

2.1. A **compact interface** also speeds up development process, because the entire interface can be remembered (after a while anyway), which reduces the reference browsing (reference is a huge time hog). To illustrate this point, here are all the 60 classes (modules) of Elgrint:

```
MVector, MVector::CIter, MVector::Iter, MHash, MHash::CIter, MMap, MMap::CIter,
MMap::Iter, MWidthList, MWidthList::CIter, MList, MList::CIter, MList::Iter,
MSortList, MSortList::CIter, MString, MImage, MImage::CIter, MImage::Iter, MRect,
MPoint, MLocus, MSize, MTransform, MChar, MColor, MFileInfo, MDriveInfo,
MDateTime, MFont, MFile, MTextFile, MApp, MSys, MPtr, MRange, MWindow,
MInfoLabel, MProgressLabel, MMultimediaLabel, MFrame, MMainFrame, MScreen,
MPageBar, MSegmentBar, MScrollBar, MDialogBox, MFileDialogBox, MButton,
MCheckButton, MDropButton, MSlideBox, MDateBox, MRangeBox, MRadioBox, MMenuBox,
MListBox, MTreeListBox, MEditBox, MHtmlBox.
```

And, yes, I just wrote them all from memory, without checking the reference even once. Needless to say, such a feat would be impossible in Qt with its 831 classes, except for a few people with phenomenal memory.

2.2. But compactness is not the only "weapon" against "time hogs" during development. Another one is the **overhead reduction**, meaning that it takes very little "meaningless" code to create app elements. Practically every programming language has certain constructs, which are needed for the compiler, but are otherwise redundant. Elgrint strives to minimize such overhead, and with great success. For example, it can take only two relatively short lines of code to create a basic Elgrint app:

```
#include <elgrint.cpp>
void MAppMain() { MMainFrame().runMessageLoop(); }
```

That's it! Other toolkits may require you to write many more lines of code (about 40 in Windows API) to accomplish the same task. And it's just one example.

2.3. Yet another "weapon" is the **optimal naming**, which makes automatic completion mechanisms (e.g. Microsoft Intellisense) more efficient. For example, the MWindow class contains a set of 9 "digger" functions, which are used quite frequently during message processing: `digKeys, digStringRef, digTimerID, digRectRef, digNN, digNID, digDir, digMessageCode, digOrigin.`

Notice that for the most part you only need to enter "dig" and one more letter for the auto-complete to work. The only exceptions are the digNN and digNID functions, where you need to enter two letters after "dig". This principle of **minimal common prefix** is used across Elgrint (wherever practical), and saves a lot of typing over time. It is just one example of many benefits that the optimal naming provides. Another benefit, as already mentioned, is that it makes remembering function names simpler.

(3). Elgrint saves debugging time, reduces frustration, and prevents many crashes.

Elgrint is designed to prevent and handle many common programming mistakes, using various techniques and principles.

3.1. One such principle is the complete **encapsulation of dynamic memory management.**
Every program needs to allocate (reserve) blocks of computer memory to store data. This memory can be accessed through the so called memory pointer variables (or simply pointers). But afterward this allocated memory must be released (unreserved), or it will not be usable anymore, and eventually the program will fill the entire memory and crash. This problem is called a "memory leak". And even if you remember to release the memory, all the pointers for this memory become invalid, and accidentally using them afterward will crash the program as well. This problem is called "garbage pointer" or "dangling pointer". These problems are particularly bad in C++, but Elgrint solves them all by making the use of pointers all but redundant.

In fact, **there is not a single pointer in Elgrint's public interface** (except the conversion from a string literal to MString, but that's transparent to the programmer), and the very use of pointers is actively discouraged (for example, Elgrint forbids converting regular pointers to window pointers).
Conversely, many toolkits - Qt in particular – require, or at least encourage, the use of regular pointers.

Removing the need for pointers is achieved by a set of customized data structures with smart iterators (iterators are basically data pointers, but in Elgrint they are much more than that), and a special class template called a "window pointer" (MPtr). A window pointer allocates window objects, and behaves like a regular pointer, but it has none of the problems of regular pointers: no memory leaks and no dangling. Even dereferencing a null window pointer (using non-assigned pointer with meaningless value), which would normally crash a program, returns a reference to a closed (non-existing) dummy window. Closed windows safely ignore any operation on them, so they can be used freely without causing any problems. In fact, using null pointers can often make the app simpler (less conditions to check).

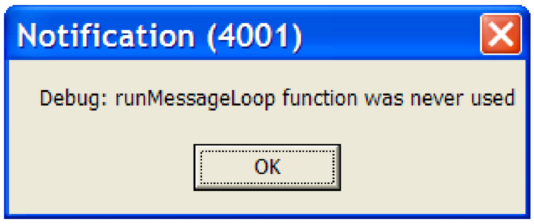
3.2. The iterators can compensate for many mistakes as well. One common mistake is to change a data structure while an iterator points to it. Normally, an iterator becomes invalid in such a case, and using it may cause unpredictable results. But Elgrint iterators detect such cases and re-validate themselves. Re-validation is fully automatic and transparent to the programmer. Even if re-validation is impossible, using an invalid iterator causes an orderly program termination – never an unpredictable behavior.
In fact, in Elgrint (as opposed to most other toolkits) **there is no such thing as "undefined behavior"** – all operations are always well-defined for any input, unless a serious system error occurs, of course.
Many modern toolkits have smart iterators, but none seem to be as safe as Elgrint in this respect, and even partial safety comes with heavy performance penalties, whereas Elgrint's iterators are not only completely safe, but **work practically as fast** as regular, unsafe pointers.

3.3. Another safety technique that reduces debugging time is the built-in error checking mechanism, which detects many common mistakes and alerts the programmer if it encounters such a mistake (mostly in debugging mode). For example, if a message handler is used outside of its proper context, it will generate a warning message explaining what happened. By the way, using invalid iterators (see 3.2 above) also generates informative errors. And so on.

Example:
Let's say you wrote the basic app (see 2.2 above), but forgot to call the *runMessageLoop* function (it does happen sometimes even to the best programmers):

```
#include <elgrint.cpp>
void MAppMain() { MMainFrame(); }
```

Normally, this app would terminate immediately (because there is nothing to run), and you will spend some time trying to figure out why nothing happens. But in Elgrint the following thing happens if you try to run this code:


Notification (4001)

Debug: runMessageLoop function was never used

Now, isn't this convenient?

(4). Write once, compile anywhere – no need to develop the same app many times.

Being a cross-platform toolkit (at least potentially) Elgrint promises a seamless portability. In other words, you write the code only once, and get apps for multiple platforms simply by compiling this code in different environments. Moreover, Miranor provides a Cloud Compilation Service, so you could compile your app for multiple platforms with a push of a button, without installing anything.

Of course, all cross-platform toolkits provide portability, but it's hardly perfect, or, as professionals say, "platform-agnostic". For example, Qt includes many functions that only work on specific platforms, so parts of a Qt-based app still need to be rewritten for different platforms.

Elgrint, on the other hand, is designed at a higher abstraction level, where the differences between platforms become unimportant. For example, Elgrint defines a concept of a "click event", which can be triggered by pressing a left mouse button or by pressing a touch-screen. Elgrint apps make no distinction between these events, thereby being compatible with both desktop computers and mobile devices.
Also, Elgrint interface is 100% ANSI-compatible, which means that it can be used with any standard compiler. Qt interface, on the other hand, contains some non-standard parts, which can only be compiled by certain compilers, thereby constraining developers even further. The same is reportedly even truer with Google's Android SDK, which is not based on standard Java, and needs Google's Eclipse compiler.

Moreover, Elgrint's compactness plays a significant role even here. Platforms come and go. Several years ago Symbian was the most popular mobile operating system, but now it's all but extinct. Just a couple of years ago, the mobile market was ruled by iOS, and now the Android is in the lead. And what will happen a few years from now? Or even next year? Will Android be superseded by Windows Blue (the latest "secret weapon" from Microsoft), or Ubuntu, or Samsung Bada, or some other platform no one has heard of yet? Most likely, yes. Worse yet, most platforms, Android in particular, are themselves fragmented into different versions, which may not be compatible with each other.
Whenever a new platform or a new version of existing one comes along, all the toolkits have to be adjusted to the change (which often means partly or even fully rewritten). But that takes time and the larger the toolkit, and the more platform-dependent it is, the more time and effort it will take, while unfortunate developers sit and wait helplessly until it's done. For example, it took Digia over a year to adapt Qt to iOS and Android, and even then the adaptation was incomplete.

Elgrint is not only highly compact, but expressly designed to be nearly platform-independent even internally. This means that only a small portion of Elgrint (some 7000 lines of code) need to be adapted to any new platform. Therefore, there is a good chance that whenever a new platform appears, Elgrint will be the first to adapt to it, and Elgrint developers will have working apps for the new platform before anyone else does.

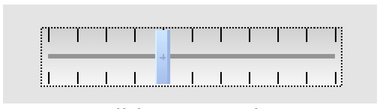
(5). Elgrint apps run fast and require less power and less memory.

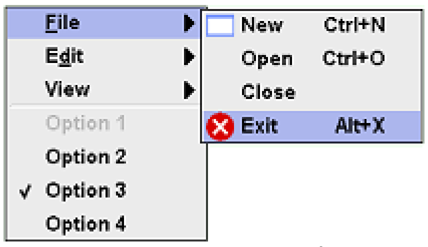
Probably the greatest benefit of C++ is its efficiency. Other things being equal, apps made with C++ run faster and require less power (in particular, prolong the battery life). Both factors are especially important for mobile devices, with their relatively weak processors and small batteries.
However, Elgrint goes one step further, by making performance a high priority. Elgrint's data structures are more efficient than the standard ones in every way (work faster and require less memory), and support a universal fast-copy mechanism, which significantly reduces the time required to pass large amounts of data between functions without using the unsafe pointers or inconvenient (and still not completely safe) references.
The messaging system is also heavily optimized, and can process about a million messages per second even on a lowest-end modern processor.

(6). Elgrint allows app developers to make money as a side effect of the development.

Most toolkits provide a fixed number of "widgets" – graphic user interface elements, e.g. button, slider, menu:

Button example


Slider example



Menu example

But if you need additional widgets, you have to make them yourself. Some open-source toolkits have communities, which share widgets between them for free, but that's usually too chaotic, and not very productive, and, most importantly, raises an important question for each developer: "What's in it for me?"

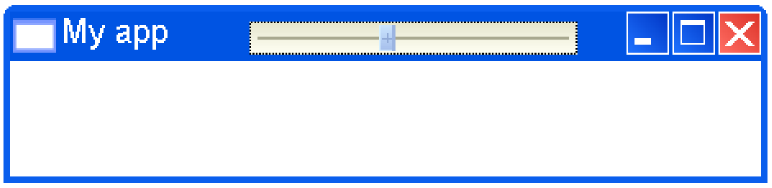
Miranor provides a **"Widget Store"** service, to which developers can submit their custom widgets, and **receive large percentages** from each sale of their widgets to other developers.

A similar model has been used with amazing success in Apple App Store and Google Play, which make billions per year by selling ready-made apps for an average price of just a few dollars. Photo-stocks, such as ShutterStock (www.shutterstock.com) use a similar model to sell images, and make nice profits as well. Selling widgets (app components) instead of complete apps or pictures should be just as successful, and not just for the provider, but for the developers as well.

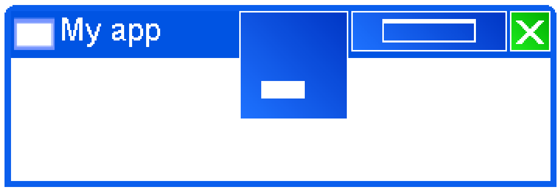
As a developer you need to create custom widgets anyway and spend time and money doing it.
Now, imagine that you can get some of this money back simply by uploading your existing work to the Miranor Store, then sit back and **watch the money** trickle back to you.
By the way, I did it with ShutterStock (just to test the idea). I created 10 decent photographs and uploaded them to the stock, and now I make a sale every few days. Yes, it's not a lot of money (it could be more if I wanted to), but it takes me zero effort to make it, so it’s well worth it.
Very few app development toolkits provide a similar Store option, and none seem to be good at it.

(7). Elgrint allows design freedom and flexibility.

7.1. Many toolkits put constraints on the structural design of apps, but not Elgrint. For example, in Elgrint you can put a slider on the title of the app frame:

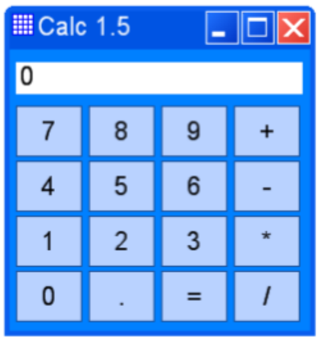


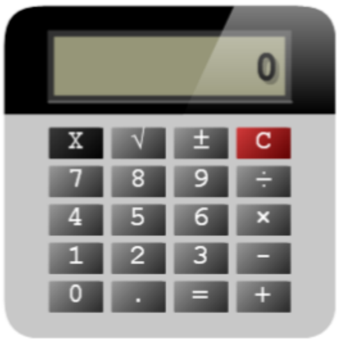
You can also customize the existing frame buttons (change size and color, for example):



Good luck trying to do any of this in Windows API or many other toolkits – it simply wouldn't work.

In fact, in Elgrint you can redesign anything you want any way you want it. Here's another example – two different designs of a simple calculator app:





Standard (predefined) design   Custom, "pro-like" design (about an hour of extra work)

Notice that the "pro" version (on the right) doesn't even have a visible title area (although it's still there), and its x-button (which closes the app, the one with the "X" in it) is moved below with the rest of the calculator's buttons (the button in the top-left corner).

But the best part is that you can **change almost any widget parameter after its creation with no extra effort.** Remember those annoying Windows messages: "To apply the changes you need to restart the program/computer".
Oh no, you don't. Not with Elgrint.